JUNIT 5 PREVIEW 38 | GENERICS IN DEPTH 45 | CEYLON LANGUAGE 50

MAY/JUNE 2016

Java™ magazine

By and for the Java community

FROM BIG DATA TO INSIGHTS

14
APACHE SPARK: GETTING STARTED

20
SPARK FOR PERSONAL PROJECTS

26
100 GB OF DATA IN MEMORY BUT OFF HEAP

ORACLE®

//table of contents /

EDITORIAL

Editor in Chief
Andrew Binstock

Managing Editor
Claire Breen

Copy Editors
Karen Perkins, Jim Donahue

Technical Reviewer
Stephen Chin

DESIGN

Senior Creative Director
Francisco G Delgadillo

Design Director
Richard Merchán

Senior Designer
Arianna Pucherelli

Designer
Jaime Ferrand

Senior Production Manager
Sheila Brennan

Production Designer
Kathy Cygnarowicz

PUBLISHING

Publisher
Jennifer Hamilton +1.650.506.3794

Associate Publisher and Audience Development Director
Karin Kinnear +1.650.506.1985

Audience Development Manager
Jennifer Kurtz

ADVERTISING SALES

Sales Director
Tom Cometa

Account Manager
Mark Makinney

Account Manager
Marcin Gamza

Advertising Sales Assistant
Cindy Elhaj +1.626.396.9400 x 201

Mailing-List Rentals
Contact your sales representative.

RESOURCES

Oracle Products
+1.800.367.8674 (US/Canada)

Oracle Services
+1.888.283.0591 (US)

ARTICLE SUBMISSION

If you are interested in submitting an article, please email the editors.

SUBSCRIPTION INFORMATION

Subscriptions are complimentary for qualified individuals who complete the subscription form.

MAGAZINE CUSTOMER SERVICE

java@halldata.com **Phone** +1.847.763.9635

PRIVACY

Oracle Publishing allows sharing of its mailing list with selected third parties. If you prefer that your mailing address or email address not be included in this program, contact Customer Service.

Java Magazine is published bimonthly and made available at no cost to qualified subscribers by Oracle, 500 Oracle Parkway, MS OPL-3A, Redwood City, CA 94065-1600.

XRebel

The Miserable Business of Hosting Projects

With more companies leaving the business and the survivors in an intense price war, is the model of free open-source hosting sustainable?

In late April, Oracle announced that it would be shuttering project-hosting sites Project Kenai and Java.net at the end of April 2017. The reason was that there was no desire to compete with sites such as Bitbucket and GitHub, among others. The closures continue a long-standing phenomenon: the exiting from open source project hosting by companies whose principal business is not closely tied to hosting code.

Project hosts did not really begin in earnest until the open source movement took root in the late 1990s.

By 1999, two principal models for open source project hosts began to emerge: the open-to-everyone sites and the more selective hosts. The latter group included Codehaus, Tigris.org, and other sites that required approval before a project could reside there. Typically, the requirements focused on the kind of license, the seriousness of the project, and whether it had the potential to catalyze a developer community.

These sites were viewed as the elite stratum. They hosted vetted projects that were likely to succeed. This model worked surprisingly well. Codehaus became the home of Groovy, Maven, Sonar, and most of the early IoC (inversion of control) frameworks—not bad for a site hosting a few hundred projects. The Apache Software Foundation and the Eclipse Foundation today pursue a similar model (although with important

PHOTOGRAPH BY BOB ADLER/GETTY IMAGES

structural differences). For this model to truly work, the sponsoring organization must be able to successfully solicit funds to cover costs and recruit volunteers to run operations. Without this kind of funding support, most of these curation-based sites have atrophied or disappeared altogether.

Facing off against them are hosts that accept all projects. For much of the previous decade, the leader was undeniably SourceForge. If curating hosts were the cathedrals, then SourceForge was the bazaar: active, noisy, filled with large amounts of low-value projects—projects abandoned immediately after setting up the site, classroom projects, and so on—interspersed with occasional jewels.

The success of this model inspired competitors—notably Google Code, which quickly became *the* place to go for developer-oriented projects. And the model sprouted a hybrid approach in which sites welcomed projects if they fulfilled some minor criteria. Java.net was such a site, with the requirement that projects be written in Java. Similar language-specific sites, such as RubyForge, followed this approach.

Competition among hosts, however, created a double burden. Not only were hosts obliged to bear the costs of providing services for free, but they also needed to regularly upgrade their offerings. Most sites, after several rounds of investing in new features, decided to throw in the towel. Google Code, Java.net, Project Kenai, JavaForge, and others have closed or are in the process of shutting down.

Part of the pressure came from new companies that have a true commercial stake in hosting projects and are willing to make continuous investments in the services: principally, Bitbucket (part of Atlassian), GitHub, and GitLab.

Their offerings are polished and deep—websites, wikis, code review tools, and defect trackers, in addition to SCM. (Extensive as these offerings are, I should point out, they are not as complete as early sites, such as Codehaus, which also offered email and mailing lists, hosted user forums, provided continuous integration, and arranged for free licenses to commercial development tools.)

While the new market leaders have earned their places through admirable products and wise community development, it's still difficult to tell whether the model of hosting huge numbers of projects at no cost can be sustained long-term. GitHub, for example, has revamped its pricing due to head-to-head competition with GitLab—with each side progressively offering more unlimited features at no cost. Obviously, that's a model that cannot continue indefinitely.

This situation is somewhat reminiscent of where publishing was five or six years ago—a time when sites competed by offering ever deeper and more-elaborate content at no cost. Eventually, the model had to self-correct, and now paid subscriptions are emerging as the new norm.

I expect that given the long list of companies exiting project hosting and the intense competition among the survivors, the model will eventually need to evolve to one in which developers pay for some of the services they now get for free. Given the high quality of the current offerings, it seems fair to me to ask that we shoulder some of those costs.

Andrew Binstock, Editor in Chief
javamag_us@oracle.com
@platypusguy

MARCH/APRIL 2016

A Note on Annotations

In "Annotations: An Inside Look" (March/April 2016, page 35), Cédric Beust wrote, "Since 2004, there has been only one major update to annotations in the JDK: JSR 308, which added more locations where annotations could be placed."

However, it's worth knowing that Java 8 improves brevity by allowing multiple annotations of the same type to be written on an element without the need for a "wrapper" annotation. For example, on page 47 of "What's New in JPA: The Criteria API," the code

```
@NamedQueries({
    @NamedQuery(name=..., query=...),
    ... })
```

could drop `@NamedQueries` if `NamedQuery` was repeatable, and simply use multiple `@NamedQuery` annotations directly on the class. The javadoc for `java.lang.reflect.AnnotatedElement` has more info for interested readers.

—Alex Buckley
Specification Lead, Java Language and VM, Oracle

var vs. val

Re: your editorial on JDK Enhancement Proposal 286 (March/April 2016, page 3), I don't agree that

```
var surprise = new HadynSymphony();
```

is better than

```
HadynSymphony surprise = new HadynSymphony();
```

In the example you provided, you didn't really cut down on the verbosity in a meaningful way; you just replaced one token with another that's shorter but also less meaningful. You went from "here's a new variable with the specified type" to "here's a new variable." Yes, the first approach potentially saves me a few keystrokes, but for whom are those keystrokes really a problem? I've been coding in Java awhile and never thought (or heard anyone else say), "Gee, I wish I didn't have to define the type of a local variable—that would save me so much time and make debugging easier."

Also, many type names are shorter—sometimes significantly so—than `HaydnSymphony`. In fact, I suspect that the somewhat longish name was deliberately chosen to exaggerate the supposed benefits of the proposal. Plus there's also the fact that something like this is fairly common:

```
Symphony surprise = new HadynSymphony();
```

Using the `var` approach here would not only eliminate useful information (the type I really care about versus the implementation that was instantiated), but all I'd get in exchange for the loss of readability is the time I saved not typing a whopping five characters. And that, in a nutshell, seems to be the problem with this proposal: the loss of code readability isn't offset by a comparable gain in the amount of time it takes to create or maintain the code. It seems like a solution in search of a problem and a desperate attempt to find some way to change Java rather than something that will actually be beneficial for either beginning or experienced programmers.

Regarding `val` versus `const`, I have to disagree there, too. In all honesty, when I first saw your `val` example, I wondered what it was that identified it as a constant in `val normalTemp = 98.6;`.

Then I realized that the earlier example (`HaydnSymphony`) used `var` while this one was using `val`. In other words, without careful scrutiny it's easy to get them confused. Part of the problem is the visual similarity, but it's also an issue of semantics: the name `val` by itself doesn't imply something that's constant. From a Java perspective, `var` doesn't just mean something that can vary, because both officially (that is, in the *Java Language Specification*, which talks about "constant variables") and informally (in your editorial), *variable* is used as an umbrella term that includes both constants and true variables. That's also how it's used in the vernacular by Java programmers, making `var` a poor choice for indicating something that's mutable and nonfinal.

The one thing I do think would be an improvement is something that wasn't even presented, specifically something like this:

```
const normalTemp = 98.6;
```

This provides the abbreviated grammar without the vagueness of `val` and seems like a no-brainer when compared to its current equivalent:

```
final float normalTemp = 98.6f;
```

Here the `const` keyword stands in for `final`, and it indicates that the type is to be inferred from the assigned value. There's still a loss of explicit information (the variable being defined is a `float`), but at least now you've gone from two tokens down to one instead of trading one token for a different and less meaningful one.

—Brett Spell

Editor Andrew Binstock responds: Thank you for your very thoughtful, detailed note. You make some excellent points, even though my preferences run counter to yours. However, I do disagree with your dismissal of the value of fewer keystrokes. You see little benefit as the initial programmer. However, as a reader of code, being faced with long listings that repeat unneeded type information is a tiring exercise that delivers little value. While you view my example as purposely lengthy to help make my point, my concern at the time was that it was unrealistically short. Developers writing enterprise apps, especially ones that rely on frameworks, are familiar with (and tired of) dealing with data types that carry much longer names. For example, from the Spring framework: `AbstractInterceptorDrivenBeanDefinitionDecorator`*. That, I grant you, is perhaps an extreme case; however, long class names are the norm in those applications.*

Shorten from the Other End

Regarding your editorial in the March/April issue about the proposed additions to Java, why is the variability removed from the left side to be inferred from the right to mimic dynamically typed language syntax? Consider your example:

```
var surprise = new HaydnSymphony();
```

versus my recommendation:

```
HaydnSymphony surprise = new();
```

I prefer the fact that a statically typed language has a strong reference to the type a variable is declared to be. The verbosity comes in when you have to repeat it for the `new` operator. My approach also makes it easier in debugging, and not just in local scope use.

Similarly for the native or fixed-value types, I am less conflicted but still prefer the type information to be added to the variable declaration:

```
val normalTemp = 98.6;
```

would convey more information if it used the literal specifier of `f`:

```
const normalTemp = 98.6f;
```

From South Africa, thanks for a wonderful magazine.

—Rudolf Harmse

Why Not Left-Side "new"?

Instead of the `var` keyword, I propose an idea I submitted a long time ago to Project Coin. It's the use of `new` on the left side of assignments. Instead of

```
HaydnSurprise surprise = new HaydnSymphony();
```

I suggest

```
new surprise = HaydnSymphony();
```

This avoids an additional keyword and is the simplest statement.

As for `val`, I think using the reserved `const` keyword is a better solution, if in fact it's truly a final constant and not just an initial assignment. I think more Java programmers come from the C and C++ languages than from Scala, although for younger programmers, Java or JavaScript is now their initial language. In any case, there's less need for simplifying constant declarations than object declarations.

—Barry Kleinman

Short Books for Learning Java

I'm contemplating becoming a mobile game developer, and I'm struggling to grasp Java as a first-time programming language. Why is it so difficult to learn? The books that I have been studying are more than 1,000 pages long. It's a serious commitment to make. Are most professional Java programmers self-taught? How should I train my mind for this? What should I be thinking of?

—Frederick Piña

Editor Andrew Binstock responds: Only a fraction of programmers are formally trained. Many are self-taught, and even those who are formally trained eventually need to be self-taught to keep up with advances in software development. When learning a new language, a common approach is to begin by writing small programs that do simple tasks. A classic, for example, is a program that accepts a centigrade temperature on the command line and displays the equivalent Fahrenheit temperature. If this approach of learning by doing—using small, incremental programs—appeals to you, then I recommend Murach's Beginning Java. *It comes in two flavors depending on which development environment you use—Eclipse or NetBeans—and it weighs in at well under 1,000 pages. I reviewed this book in the January/February 2016 issue, on page 12.*

Contact Us

We welcome comments, suggestions, grumbles, kudos, article proposals, and chocolate chip cookies. All but the last two might be edited for publication. If your note is private, indicate this in your message. Write to us at javamag_us@oracle.com. For other ways to reach us, see the last page of this issue.

//events /

PHOTOGRAPH BY DIEGO TORRES SILVESTRE/FLICKR

JavaOne Latin America *JUNE 28–30*
SÃO PAULO, BRAZIL
The Latin American version of the premier Java event includes presentations by the core Java team, tutorials by experts, and numerous information-rich sessions, all focused tightly on Java.

GeeCON
MAY 11–13
KRAKOW, POLAND
GeeCON is a conference focused on Java and JVM-based technologies, with special attention to dynamic languages such as Groovy and Ruby. The event covers topics such as software development methodologies, enterprise architectures, design patterns, and distributed computing. More than 80 sessions are slated.

O'Reilly OSCON
MAY 16–19
AUSTIN, TEXAS
The popular open source conference moves to Texas this year, with two days of training and tutorials before the two-day conference. Topics this year include Go unikernels, scaling microservices in Ruby, Apache Spark for Java and Scala developers, and an Internet of Things (IoT) keynote from best-selling science fiction author and Creative Commons champion Cory Doctorow.

JavaCro
MAY 18–20
ROVINJ, CROATIA
JavaCro, hosted by the Croatian Association of Oracle Users and Croatian Java Users Association, will once again be held on St. Andrew Island, also known as the Red Island. Touted as the largest Java community event in the region, JavaCro is expected to gather 50 speakers and 300 participants.

JEEConf
MAY 20–21
KIEV, UKRAINE
JEEConf is the largest Java conference in Eastern Europe. The annual conference focuses on Java technologies for application development. This year offers five tracks and 45 speakers on modern approaches in the development of distributed, highly loaded, scalable enterprise systems with Java, among other topics.

jPrime
MAY 26–27
SOFIA, BULGARIA
jPrime is a relatively new conference with talks on Java, various languages on the JVM, mobile and web development, and best practices. Its second edition will be held in the Sofia Event Center, run by the Bulgarian Java User Group, and backed by the biggest companies in the city. Scheduled speakers this year include former Oracle Java evangelist Simon Ritter and Java Champion and founder of JavaLand Markus Eisele.

IndicThreads
JUNE 3–4
PUNE, INDIA
IndicThreads enters its 10th year featuring sessions on the

latest in software development techniques and technologies, from IoT to big data, Java, web technologies, and more.

Devoxx UK
JUNE 8–10
LONDON, ENGLAND
Devoxx UK focuses on Java, web, mobile, and JVM languages. The conference includes more than 100 sessions, with tracks devoted to server-side Java, architecture and security, cloud and containers, big data, IoT, and more.

JavaDay Lviv
JUNE 12
LVIV, UKRAINE

PHOTOGRAPH BY DAVID HOLT/FLICKR

More than a dozen sessions are planned on topics such as Java SE, JVM languages and new programming paradigms, web development and Java enterprise technologies, big data, and NoSQL.

Java Enterprise Summit
JUNE 15–17
MUNICH, GERMANY
Java Enterprise Summit is a large enterprise Java training event held in conjunction with a concurrent Micro Services Summit. Together, they feature 24 workshops covering topics such as the best APIs, new architecture patterns, JavaScript frameworks, and Java EE. (No English page available.)

JBCNConf
JUNE 16–18
BARCELONA, SPAIN
The Barcelona Java Users Group hosts this conference dedicated to Java and JVM development. Last year's highlights included tracks on microservices and Kubernetes.

The Developer's Conference (TDC)
JULY 5–9
SÃO PAULO, BRAZIL
Celebrating its 10th year, TDC is one of Brazil's largest conferences for students, developers, and IT professionals. Java-focused content on topics such as IoT, UX design, mobile development, and functional programming are featured. (No English page available.)

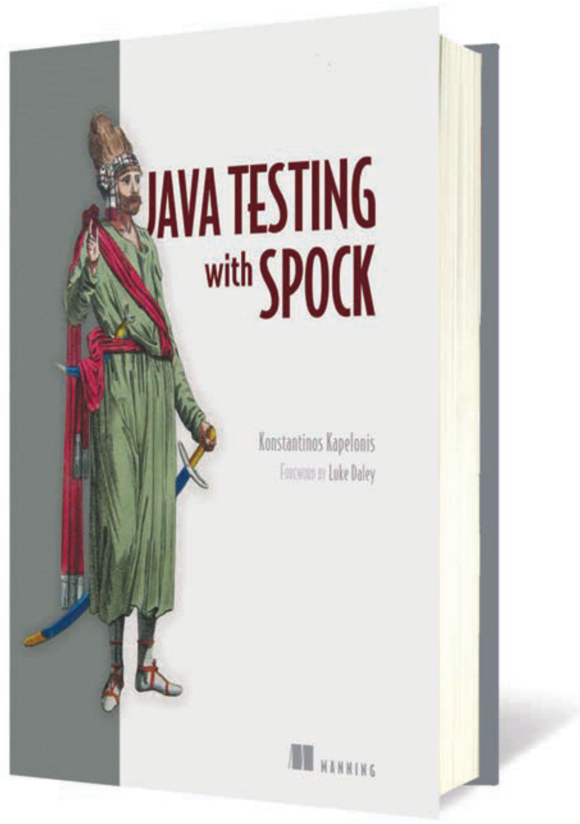
Java Forum
JULY 6–7
STUTTGART, GERMANY
Organized by the Stuttgart Java User Group, Java Forum typically draws more than 1,000 participants. A workshop for Java decision-makers takes place on July 6. The broader forum will be held on July 7, featuring 40 exhibitors and including lectures, presentations, demos, and Birds of a Feather sessions. (No English page available.)

JCrete
JULY 31–AUGUST 7
KOLYMBARI, GREECE
This loosely structured "unconference" will take place at the Orthodox Academy of Crete. A JCrete4Kids component introduces youngsters to programming and Java. Attendees often bring their families.

JavaZone
SEPTEMBER 7–8
OSLO, NORWAY
This event consists of a day of workshops followed by two days of presentations and more workshops. Last year's event drew more than 2,500 attendees and featured 150 talks covering a wide range of Java-related topics.

JavaOne
SEPTEMBER 18–22
SAN FRANCISCO, CALIFORNIA
The ultimate Java gathering, JavaOne features hundreds of sessions and hands-on labs. Topics include the core Java platform, security, DevOps, IoT, scalable services, and development tools.

Send us a link and a description of your event four months in advance at javamag_us@oracle.com.



JAVA TESTING WITH SPOCK

By Konstantinos Kapelonis
Manning Publications

The Spock testing framework has been around for many years. It was used originally by developers who wanted to leverage Groovy's uniquely good support for scripting unit tests. Over its lifetime, it has evolved into a comprehensive framework that includes a unit-test runner (which drives JUnit), a mocking framework, and a behavior-driven development (BDD)-style testing tool. As a result, you can add Spock to your testing without the need for separate mocking/stubbing tools or behavior-testing add-ons.

I first started using Spock years ago because of its excellent support for data-driven tests (a leadership position it still retains). Since then, my admiration has only grown. The big knock on it during the intervening years was the need to learn Groovy scripting to write tests. But as Groovy has grown in popularity (driven in good part by Gradle), this design is seen increasingly as an advantage.

Correctly written, Spock tests can work at the unit level (for test-driven development, for example), for integration tests, for doing BDD, and for system-wide tests. Spock is one of the few tools that can be used by both developers and QA teams, without the excess complexity such a product would imply.

What Spock has lacked, however, is good documentation. Even today, its website is deficient in many ways (despite active product development and fairly active mailing lists). This book fills that gap. It provides what you need to get started (including integration with IDEs and build tools) and then pushes onward into exploiting Spock's features, both its principal capabilities and the advanced options. To overcome the language barrier for developers mostly familiar with Java, Konstantinos Kapelonis provides a 30-page introduction to Groovy that presents the basic knowledge needed to write and run tests, even complex ones.

The author's style is clear and rarely lacking. In addition, the examples he provides display his deep familiarity with Spock and the role it plays in modern development. Although the book is written for developers, Kapelonis dips into QA perspectives on testing for the issues raised. He also combines BDD with mocking from which he elaborates an interesting mock-based approach to design—all of which gives this book a commendable applicability. Recommended.
—*Andrew Binstock*

From Big Data to Insights

While batch processing of data has been part of enterprise computing since its earliest days, so-called *big data* brings to it the benefits of considerable scale, good performance, and the ability to investigate data deeply on comparatively inexpensive hardware. The core change that makes this wave of processing innovation possible is the software that can exploit runtime advances. Not so long ago, the software center of the big data universe was Apache Hadoop. Two years later, other packages such as Apache Spark extend Hadoop's original mission. Our first feature article (page 14) explains how Spark works and how comparatively easy it is to understand and use. The second article on Spark (page 20) is for nonenterprise developers and hobbyists who want to try out big data on smaller projects.

However, Spark is not the only approach to massive data sets. Sometimes, you need to do things the old-fashioned way. Our article on JDBC for large data volume (page 30) gives handy reminders for not overtaxing the database server. And our final article (page 26) explains how one company designed and built a massive in-memory, off-heap queue. It's open source and written strictly in Java, and it stores tens of gigabytes outside the JVM. (In fact, JVMs can share data through this queue.)

There are surely more ways than these to be part of the big data revolution, but these will get you started and enable you to do some seriously fun experimenting.

ART BY WES ROWELL

Apache Spark 101

Getting up to speed on the popular big data engine

DIANA CARROLL

In recent years, the amount of data that organizations process has grown astronomically—as much as hundreds of terabytes a day in some cases, and dozens or even hundreds of petabytes in total. This "big data" far exceeds what can be stored or processed on a single computer. Handling the volume, velocity, and variety of data required by modern applications has prompted many organizations to move to distributed systems, where clusters of dozens or hundreds of computers work together to solve their data ingestion, processing, and analysis needs.

But distributed programming is challenging: the complexity involved with keeping data and processes in sync while dealing with the reality of limited bandwidth and individual system failures initially meant programmers were spending more time on the plumbing of distributed systems than actually processing or analyzing their data.

One of the most successful approaches to solving these issues has been Apache Hadoop and its principal core components: the Hadoop MapReduce data processing engine and the Hadoop Distributed File System (HDFS) data storage platform. Hadoop has been widely adopted and is used today by many organizations. But Hadoop MapReduce has limitations: it is cumbersome to program; it fully supports only Java (with limited support for other languages); and it is bottlenecked by the requirement that data be read from disk and then written to disk after each task.

Apache Spark is designed as the next-generation distributed computing framework, and it takes Hadoop to the next level.

What Is Spark?

Spark is a fast, scalable, general-purpose distributed processing engine. It provides an elegant high-level API for in-memory processing and significant performance improvements over Hadoop MapReduce.

Spark includes not only the core API but also a rich set of libraries, which includes Spark SQL for interacting with structured or tabular data; Spark Streaming for processing streaming data in near real time; MLlib for machine learning; and GraphX for graph processing.

Spark is written in Scala, a scalable JVM language with a Java-inspired syntax. In addition to Scala, the Spark API also supports Java, Python, and R, making it easy to integrate with third-party libraries and accessible to developers with a wide range of backgrounds.

Spark was originally conceived and developed at Berkeley's AMPLab. Now, it is an Apache project and is available directly from Apache or preintegrated with several Apache Hadoop distributions including those from Cloudera and other vendors.

Spark Architecture Overview

Although Spark can be run locally on a single computer for testing or learning purposes, it is more often deployed on a distributed computing cluster that includes the following:

- **A distributed data storage platform.** This is most often HDFS, but Spark is increasingly being deployed on other distributed file storage systems such as Amazon Simple

Storage Service (S3). As Spark emerges as the new default execution engine for the Hadoop ecosystem, support for it is increasingly included in new projects such as Apache Kudu (an Apache incubator project).

- **A cluster resource management platform.** Hadoop YARN (which is part of core Hadoop) is the most common such platform for enterprise production systems. Spark also includes a small built-in cluster system called Spark Standalone, which is suitable primarily for small clusters, testing, or proof-of-concept deployment. Spark supports Apache Mesos as well. Because YARN is the most common, I have chosen to use it with the examples for this article, but the concepts are similar for all three supported cluster platforms.

Spark can be invoked interactively using the Spark shell (available for Scala or Python), or you can submit an application to run on the cluster. Both are shown as options in Figure 1. In both cases, when the application starts, it connects with the YARN Resource Manager. The Resource Manager provides CPU and memory resources on worker nodes within the cluster to run the Spark application, which consists of a single *driver* and a number of *executors*.

The driver is the main program, which distributes and manages tasks that run on the executors. The driver and each executor run in their own JVM running on cluster worker nodes. (You can also configure your application so that the driver runs locally rather than on the cluster, but this is less common in production environments.)

Code Examples

To help demonstrate how the Spark API works, I walk through two code examples below. While Spark does support Java, the vast majority of installations use Scala. The examples are written in Scala and run in the interactive Spark shell. If you don't know Scala, don't worry; the syntax is similar to Java and my explanations will make the function of the code clear.

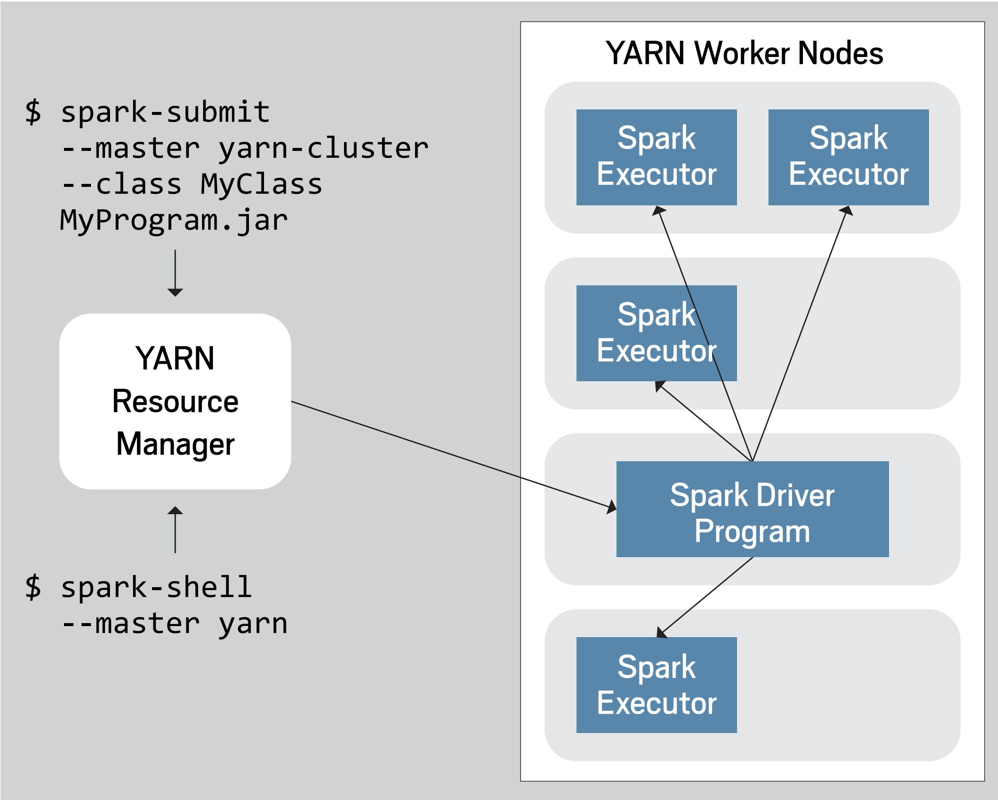


Figure 1. Spark and YARN working together

If you want to work through the examples on your own, you first need to download and run Spark. The easiest approach is to download the "Pre-built for Hadoop 2.6 and later" package of the latest release of Spark and simply unpack it into your home directory. Once it is unpacked, you can run the `spark-shell` script from the package's bin directory. For simplicity, these instructions start the shell locally rather than on a Hadoop cluster. Detailed instructions on how to download and launch Spark are on the Spark website.

You also need to download and unpack the example data, from the *Java Magazine* download area. The code examples assume that you have unpacked the data directory (`weblogs`) into your home directory.

Example 1: Count Unique Visitors

The example data set is a collection of web server log files from the customer support site of a fictional mobile provider

called Loudacre. This is a typical line from the sample data (wrapped here to fit):

```
3.94.78.5 - 69827 [15/Sep/2013:23:58:36 +0100]
  "GET /KBDOC-00033.html HTTP/1.0" 200 14417
  "http://www.loudacre.com"
  "Loudacre Mobile Browser iFruit 1"
```

The task in this first example is to find the number of unique site visitors—that is, the number of user IDs in the data set. User IDs are in the third field in the data file, such as `69827` in the example data above.

The Spark context. Every Spark program has a single *Spark context* object, an instance of the `SparkContext` class. When using the interactive Spark shell, this object is automatically created for you and given the name `sc`. When writing an application, you will create one yourself.

The Spark context provides access to Spark functionality, such as defining and loading data sets and configuring the application. The Spark context is the entry point for all Spark functionality.

```
scala> val weblogs =
|       sc.textFile("weblogs/*")
```

(Note that the Spark shell `scala>` prompt is shown at the beginning of each line. Line continuation is indicated with a pipe character: `|`. Do not enter the prompt or pipe character in the shell. Also, Microsoft Windows users might need to substitute the full Windows pathname of the example data directory, such as C:\\Users\\diana\\weblogs*.)

Because my code example uses the interactive Spark shell, I can use the precreated Spark context `sc`. The line of code above creates a new *Resilient Distributed Dataset* (RDD) object for the data in the specified set of files, and assigns it to a new immutable variable called `weblogs`. The files are located in your home directory in the default file system. (If you download Spark as described previously, the default file system is simply your computer's hard drive. If you install Spark as part of a Hadoop distribution, such as Cloudera's, the default file system is HDFS.)

Resilient distributed data sets. RDDs are a key concept in Spark programming. They are the fundamental unit of computation in Spark. RDDs represent a set of data, such as a set of `weblogs` in this example.

RDDs are *distributed* because the data they represent may be distributed across multiple executors in the cluster. The tasks to process that data run locally on the executor JVM where that data is located.

RDDs are *resilient* because the *lineage* of the data is preserved and, therefore, the data can be re-created on a new node at any time. Lineage is the sequence of operations that was applied to the base data set, resulting in its current state. This is important because one of the challenges of distributed computing is dealing with the possibility of node failure. Spark's answer to this challenge is RDD lineage.

RDDs can contain any type of data, including objects and nested data, such as arrays and sets. In this example, the `weblogs` RDD contains strings—each element is a string corresponding to a single line in the files from which the data is loaded.

RDDs provide many methods to transform and interact with the data they represent. Below are two simple examples: `count()`, which returns the number of items in the data set, and `take(n)`, which returns an array of the first `n` items in the data set.

```
scala> weblogs.count()
Long = 574023

scala> weblogs.take(2)
» Array[String] = [3.94.78.5 - 69827
    [15/Sep/2013:23:58:36 +0100]
    "GET /KBDOC-00033.html HTTP/1.0"
```

```
200 14417 "http://www.loudacre.com"
"Loudacre Mobile Browser iFruit 1",

19.38.140.62 - 21475
[15/Sep/2013:23:58:34 +0100]
"GET /KBDOC-00277.html HTTP/1.0"
200 15517 "http://www.loudacre.com"
"Loudacre Mobile Browser Ronin S1"]
```

The character » indicates the result of a command executed in the Spark shell. [The blank line in the output was added to highlight the two array elements. —*Ed.*]

Transformations and actions. In addition to those shown in this example, Spark provides a rich set of dozens of operations you can perform with an RDD. These operations are categorized as either *actions* or *transformations.*

Actions return data from the executors (where data is processed) to the driver (such as the Spark shell or the main program). For example, you saw above that the `count()` action returns the number of data items in the RDD's data set. To do this, the driver initiates a task on each executor to count its portion of the data, and then it adds those together to produce the final total. Other examples of actions include `min()`, `max()`, `first()` (which returns the first item from the data set), `take()` (seen earlier), and `takeSample()` (which returns a random sampling of items from the data set).

Single-item operations are powerful, but many types of data processing and analysis require aggregating data across multiple items. **Fortunately, the Spark API provides a rich set of aggregation functions.**

A transformation is an RDD operation that transforms the data in the base or *parent* RDDs to create a new RDD. This is important because RDDs are immutable. Once loaded, the data associated with an RDD does not change; rather, you perform one or more transformations to create a new RDD with the data you need in the form you need.

Let's examine one of the most common transformations, `map()`, continuing with the previous example. `map()` applies a function, which is passed to each item in the RDD to produce a new item:

```
scala> val userids = weblogs.
|        map (item => item.split(" ")(2))
```

You might be unfamiliar with the double-arrow operator (`=>`) in the call to the `map` method. This is how Scala represents lambda functions. Java 8, which also provides lambda functions, uses similar syntax with a single arrow (`->`).

In this case, the `map()` method calls the passed function once for each item in the RDD. It splits the weblog entry (a String) at each space character, and returns the third string in the resulting array. In other words, it returns the user ID for each item in the data set.

The results of the transformation are returned as a new RDD and assigned to the variable `userids`. You can take a look at the results of the new RDD by calling the action methods described above, `count()` and `take()`:

```
scala> userids.count()
» Long = 574023
```

Note that the total count is the same for the new RDD as it was for the parent (base) RDD; this is always true when using the `map` transformation because `map` is a one-to-one function, returning one new item for every item in the existing RDD.

`take(2)` returns an array of the first two items—here, the user IDs from the first two lines of data in the data set:

```
scala> userids.take(2)
» Array[String] = [69827,21475]
```

At this point, this example is almost complete; the task was to find the number of *unique* site visitors. So another step is required: to remove duplicates. To do this, I call another transformation, `distinct()`, which returns a new RDD with duplicates filtered out of the original data set. Then only a single action, `count()`, is needed to complete the example task.

```
scala> userids.distinct().count()
» Long = 12582
```

And there's the answer: there are 12,582 unique site visitors in the data set.

Note the use of chaining in the previous code snippet. Chaining a sequence of operations is a very common technique in Spark. Because transformations are methods on an RDD that return another RDD, transformations are chainable. Actions, however, do not return an RDD, so no further transformations can be appended on a chain that ends with an action, as in this example.

Example 2: Analyze Unique Visitors

I'll move on to a second example task to demonstrate some additional Spark features using *pair RDDs*. The task is to find all the IP addresses for each site visitor (that is, each unique user ID) who has visited the site.

Pair RDDs. The previous example (finding the number of distinct user IDs) involved two transformations: `map` and `distinct`. Both of these transformations work on individual items in the data set, either transforming one item into another (`map`), or retaining or filtering out an item (`distinct`). Single-item operations such as these are powerful, but many types of data processing and analysis require aggregating data across multiple items. Fortunately, the Spark API provides a rich set of aggregation functions. The first step in accessing these is to convert the RDD representing your data into a pair RDD.

A pair RDD is an RDD consisting of key-value pairs. Each element in a pair RDD is a two-item tuple. (A *tuple* in Scala is a collection similar to a Java list that contains a fixed number of items, in this case exactly two.) The first element in the pair is the *key*, and the second is the *value*. For this second example, you need to construct an RDD in which the key is the user ID and the value is the IP address for each item of data.

Spark is a powerful, high-level API that provides programmers the ability to perform complex big data processing and analysis tasks on a distributed cluster, without having to be concerned with the "plumbing" that makes distributed processing difficult.

```
scala> val userIPpairs = weblogs.
|        map(item => item.split(" ")).
|        map(strings => (strings(2), strings(0)))
```

Several things are going on in the code snippet above. First, note the two calls to `map` in the same line. This is another example of chaining, this time the chaining of multiple transformations. This technique is very common in Spark. It does not change how the code executes; it is simply a syntactic convenience.

The first `map` call splits up each `weblog` line by space, similar to the first example. But rather than selecting just a single element of the array that is returned from the split, the whole array is returned containing all the fields in the current data item. Therefore, the result of the first `map` call is an RDD con-

sisting of string arrays, one array for each item (`weblog` line) in the base RDD.

The input to the second `map` call is the result of the first `map` call—that is, the RDD consisting of string arrays. The second `map` call generates a pair (tuple) from each array: a key-value pair consisting of the user ID (third element) as the key and the IP address (first element) as the value.

Use `first()` to view the first item in the `userIPpairs` RDD:

```
scala> userIPpairs.first()
» (String, String) = (69827,3.94.78.5)
```

Note that the first item is, in fact, a pair consisting of a user ID and an IP address. The `userIPpairs` RDD consists of the same number of items as the base RDD, each paired with a single IP address. Each pair corresponds to a single line in the `weblog` file.

Data aggregation. Now that the RDD is in key-value form, you can use several aggregation transformations, including `countByKey` (which counts the number of values for each key), `reduceByKey` (which applies a function you pass to sum or aggregate all the values for a key), `join` (which groups values associated with keys from two different RDDs), and `mapValues` (which applies a function to transform the value of each key). To complete the task for the second example, use the `groupByKey` transformation, which groups all the values for each key in the RDD into a single collection:

```
scala> val userIPs=userIPpairs.groupByKey()
scala> userIPs.first()
» (String, Iterable[String]) =
  (52427,CompactBuffer(241.216.103.191,
   241.216.103.191, 70.50.111.153,
   70.50.111.153, 174.222.251.149,
   174.222.251.149, 20.23.59.237,
   20.23.59.237, 190.196.39.14,
   190.196.39.14, 67.250.113.209,
   67.250.113.209, …))
```

In the snippet above, the `groupByKey` transformation results in a new pair RDD, in which the pair's key is the same as it was before, a user ID, but the pair's value is a collection of all the values (IP addresses) for that key in the data set.

Conclusion

As you can see, Spark is a powerful, high-level API that provides programmers the ability to perform complex big data processing and analysis tasks on a distributed cluster, without having to be concerned with the "plumbing" that makes distributed processing difficult. This article has barely scratched the surface of Spark's capabilities and those of its add-on libraries, such as Spark Streaming and Spark SQL. For more information, start at the Apache Spark website. To explore Spark and the Hadoop stack independently, download the Cloudera QuickStart Virtual Machine or Docker image. </article>

Diana Carroll is a senior curriculum developer at Cloudera. She has been working with and teaching about Spark since version 0.9 in late 2013. Carroll wrote and taught one of the first commercially available courses on Java in 1997, and has also produced more than 20 courses for software developers, system administrators, data analysts, and business users on subjects ranging from ecommerce to business process management.

learn more

Wikipedia entry for Apache Spark

Another introductory article on Spark

NIC RABOY

Using Spark and Big Data for Home Projects

Create a small personal project using big data pipelines.

In every application, there is a need to move data around, and the larger the application, the more data is involved in this process. A mandatory step before using any kind of data is to prepare it. You need to clean the data by removing useless parts and then shape or structure it so that it fits your processes. This could include adding a default value for missing data, trimming the whitespace from strings, removing duplicates, or anything else. There are many things that can be done, all tied to what you want to do with the data.

Often, the format of the data you're working with is subject to change. Being able to remain flexible in the database layer is critical in these scenarios because, as a developer, you should not spend your development time maintaining database schemas. A NoSQL database is particularly helpful due to its ability to remain flexible, allowing you to focus on your code and work with data instead of worrying about how the data exists in the database.

In this article, I present a personal example. As a couple expecting their first child, my wife and I came across a situation that every expecting parent encounters: needing to pick a name for our baby. Being the software developer that I am, it made sense to write an application that could supply ideas for a name that my wife and I might both like.

A data set on the Kaggle data science website contains a list of baby names that have been chosen in the US for almost a century. This data set—one of many on the internet—greatly facilitates experimenting with big data. Although not a need for me, in many use cases, a common requirement is doing analysis in real time with a massive amount of input.

In this project, I show how to ingest this unstructured, dirty, comma-separated values (CSV) data into NoSQL using Couchbase and process it in real time using RxJava or Apache Spark, so it can later be used in my application.

The Elements of the Big Data Pipeline

As a Java developer, you'll often need to load CSV data into your database. CSV data is raw data, to an extent, and typically must first be processed to match your data model.

The baby name data that I am using is raw CSV data. There are several data files in this data set, but my needs only regard how many times each name was used every year. This information can be found in NationalNames.csv, which is structured as follows:

- ID
- Name
- Year
- Gender
- Count

Before this data can be queried and analyzed, it must first be processed into a more manageable format. There are several ways to transform it. For this example I'm using both RxJava and Apache Spark.

What is RxJava? If this is your first time hearing of Reactive Java, otherwise known as RxJava, it is defined this way in the official documentation:

"RxJava is a Java VM implementation of ReactiveX (Reactive Extensions): a library for composing asynchronous and event-based programs by using observable sequences."

Spark offers in-memory data storage and processing, which makes it significantly faster than competing big data technologies.

What does this mean? RxJava is reactive programming, which in this case means programming with asynchronous data streams. These data streams can be anything from application variables to data structures. It doesn't matter. You can listen for these streams and react appropriately when they come in.

In the example of processing a CSV file, each line of the CSV file could be a data stream. When the line is read, RxJava reacts and transforms it as necessary.

What is Apache Spark? Apache Spark is a data-processing engine designed to handle massive amounts of data. While RxJava works well, it wasn't meant to handle the petabytes of information in an enterprise application. Spark offers in-memory data storage and processing, which makes it significantly faster than competing big data technologies. I'll use it for that purpose. [An accompanying article in this issue, "Apache Spark 101" (page 14), presents a deep dive into using Apache Spark. *—Ed.*]

Before querying and analyzing the loaded and transformed data, the goal is to get the unstructured data into a NoSQL database. For this example, I'll use the NoSQL document database, Couchbase.

What is Couchbase? Couchbase is an open source NoSQL document database. Document databases such as Couchbase store complex data using JSON, the same data format commonly used in public-facing APIs. For example, the following would be considered a JSON document:

```
{
    "first_name": "Nic",
    "last_name": "Raboy",
    "address": {
        "city": "San Francisco",
        "state": "California",
        "country": "USA"
    }
}
```

Notice that nested objects and arrays are fair game in a schemaless database such as this. The benefit of using a schemaless database instead of a relational database is flexibility in how the data is stored. You don't need to declare any constraints beforehand, and if the data is irregular you can add or omit fields as needed. Raw data can be a mess, and sometimes it's easier to store it without forcing it into a relational schema.

NoSQL document databases are not the only type of NoSQL database platforms, and NoSQL is not the only type of database platform.

In the NoSQL database market segment, there are several kinds of options: document, key-value, tabular databases, and specialized products such as graph databases—all of which store data differently. Unlike relational databases, NoSQL databases store data in an unstructured way, as demonstrated in the JSON example.

Couchbase is distinguished by its memory-centric architecture. It is composed of a RAM layer and a persisted disk layer. Data is written to and read from the memory layer and asynchronously written to disk. This makes Couchbase very quick and a good resource to use with big data.

Couchbase Apache Spark Connector. Apache Spark also does

most of its data processing in memory, using hard disk only when necessary. Couchbase uses a Spark connector to move data into and out of the database directly. This uses a feature of Spark called resilient distributed data sets, most often referred to as RDDs.

Apache Spark was designed to process massive amounts of data.

Loading and Transforming the CSV Data into Couchbase

As mentioned a few times earlier, the CSV data for the baby names first needs to be loaded into a database before I start to process it. There are many ways to do this, but when working with potentially massive amounts of data, it would make the most sense to load this data either through RxJava or Apache Spark.

Coming from a Java background, you might not be familiar with big data tools such as Apache Spark, and that's not a problem. This CSV data set, with roughly 2 million records, can be loaded successfully using Java.

The requirements for loading with RxJava. There are a few dependencies that must be included in the Java project before attempting to load the CSV data into Couchbase:

- A CSV reader such as OpenCSV
- RxJava
- The Couchbase Java SDK

You can obtain all of these dependencies through Maven by including them in the project pom.xml file.

Developing an RxJava CSV loader. I'll create a class—it doesn't matter what I call it—that represents the RxJava way for processing the CSV file. I'll also create a class for the Apache Spark way later.

To load, but not read, the CSV file, I'll create a new `CSVReader` object, as follows:

```
CSVReader reader =
  new CSVReader(new FileReader("PATH_TO_CSV_FILE"));
```

Because the data will eventually be written to the database, I must connect to my server and open the bucket, which is a collection of NoSQL documents.

```
Bucket bucket =
  CouchbaseCluster.create("http://localhost:8091").
    openBucket("default", "");
```

This code assumes that Couchbase is running locally and the data will be saved in the default bucket without a password.

To process the CSV data set, I must create an RxJava `Observable`:

```
Observable
  .from(reader)
  .map(
        csvRow -> {
          JsonObject object = JsonObject.create();
          object
            .put("Name", csvRow[1])
            .put("Year", csvRow[2])
            .put("Gender", csvRow[3])
            .put("Count", csvRow[4]);
          return JsonDocument.create(
            csvRow[0], object);
        }
    )
    .subscribe(document -> bucket.upsert(document),
               error -> System.out.println(error));
```

Let's look at what is happening in the `Observable`. The `CSV Reader` creates an `Iterable<String[]>`. The `Observable` will use the `Iterable<String[]>` as the source of data for the `.from()` method.

The data that is read will be an array of strings, not something that can be stored directly in the database. Using the

`.map()` function, the array of strings can be transformed into whatever I decide. In this case, the goal is to map each line of the CSV file to a database document. During this mapping process, I could do some data cleanup. For example, while I don't do it here, I could use something such as `csvRow[*].trim()` to strip out any leading and trailing whitespace in each CSV column.

Finally, I must save to the database each read line that's processed. The `.subscribe()` method from the previous snippet of code subscribes to notifications that the `Observable` emits—in this case, the manipulated data record.

After executing the RxJava class for loading CSV data, there should be almost 2 million documents created containing baby names, containing the fields specified in the `Observable`. An example of one of the documents might be this:

```
{
    "Name": "Nicolas",
    "Year": "1988",
    "Gender": "M",
    "Count": 400
}
```

Transforming raw data into JSON using Apache Spark. RxJava works great, but what if you're working with hundreds of millions of CSV records? Apache Spark was designed to process massive amounts of data.

Using the same sample data as in the RxJava segment, I give Apache Spark a try.

The requirements for loading with Apache Spark. I must include a few dependencies in the Java project before I attempt to load the CSV data into Couchbase:

- Spark Core
- Spark CSV
- Spark SQL
- Couchbase Apache Spark Connector

You can obtain all of these dependencies through Maven by including them in your pom.xml file.

Developing an Apache Spark CSV loader. To use Apache Spark in the Java project, I must first configure it in the code:

```
SparkConf conf = new SparkConf()
   .setAppName("Simple Application")
   .setMaster("local[*]")
   .set("com.couchbase.bucket.default", "");
JavaSparkContext javaSparkContext =
    new JavaSparkContext(conf);
```

The application name will be `Simple Application`, and the master Spark cluster will be the local machine, because Spark will be running locally in this example. The Couchbase bucket I will use is once again the default bucket.

To proceed, I need to create a Spark *DataFrame*. DataFrames are distributed collections of data that are organized similarly to a relational database table and are particularly useful when it comes to CSV data because CSV data closely resembles a table.

To create a Spark DataFrame, I need to create a `SQL Context`. I can do this by using the `JavaSparkContext`. The `JavaSparkContext` represents a connection to the Spark cluster, which in this case is being run locally. With the `JavaSparkContext`, a `SQLContext` can be created that allows for the creation of DataFrames and the usage of SparkSQL:

```
SQLContext sqlContext =
    new SQLContext(javaSparkContext);
```

Using the `SQLContext`, the CSV data can be read like this:

```
DataFrame dataFrame = sqlContext.read()
    .format("com.databricks.spark.csv")
```

```
    .option("inferSchema", "true")
    .option("header", "true")
    .load("PATH_TO_CSV_FILE");
```

The read process will use the Spark CSV package and preserve the header information that exists at the top of the CSV file. When read into a `DataFrame`, the CSV data is now something Couchbase can understand. There is no need to map the data, as was done with RxJava.

I must make an adjustment to the ID data. Spark will recognize it as an integer or numeric value because this data set has only numeric values in the column. Couchbase, however, expects a string ID, so this bit of Java code using the Spark API solves the problem:

```
dataFrame = dataFrame.withColumn(
    "Id", df.col("Id").cast("string"));
```

I can now prepare the `DataFrame` for saving to the database:

```
DataFrameWriterFunctions
  dataFrameWriterFunctions =
    new DataFrameWriterFunctions(
        dataFrame.write());
Map<String, String> options =
  new Map.Map1<String, String>("idField", "Id");
```

With the `DataFrame` data piped into the appropriate `DataFrameWriterFunctions` object, I can map the ID value to a document ID. At this point, I can save the data:

```
dataFrameWriterFunctions.
    couchbase(options);
```

By calling the Couchbase function of `DataFrameWriterFunctions`, massive amounts of Couchbase documents will now be saved to the bucket.

I can execute the project after I package it by doing something like the following:

```
/path/to/apache/spark/bin/spark-submit \
--class "com.app.Main" \
target/project-jar-with-dependencies.jar
```

Querying the Data for a Perfect Baby Name

Until now, the raw CSV data containing the baby names has been transformed and saved as JSON data in the database. The goal of this project hasn't been met yet. The goal was to come up with some nice baby name options. The project is in a good position at the moment, because the data is now in a format that can be easily queried.

Choosing a great name with RxJava. With the naming data loaded into Couchbase, it can now be queried. In this instance, I'm going to use RxJava to query the data to try to come up with a good baby name.

Let's say, for example, that I want to name my baby using one of the most popular names. I could create the following RxJava function:

```
public void getPopularNames(
  String gender, int threshold) {
    String queryStr =
      "SELECT Name, Gender, SUM(Count) AS Total " +
      "FROM 'default' WHERE Gender = $1 GROUP BY " +
      "Name, Gender HAVING SUM(Count) >= $2";
    JsonArray parameters = JsonArray.create()
      .add(gender)
      .add(threshold);
    ParameterizedN1qlQuery query =
       ParameterizedN1qlQuery.parameterized(
         queryStr, parameters);
    this.bucket
      .query(query)
      .forEach(System.out::println);
}
```

In the previous function, two parameters can be passed: the gender I am looking for and a threshold to determine popularity. Let's say, for example, my wife and I find out we're having a boy (I don't know yet), and we want to choose from a list of names that have been used more than 50,000 times. I could pass `M` and `50000` into the function and the parameterized query would be executed.

Each row that satisfies the query would be printed to the console logs.

By using Apache Spark and Couchbase, **you can process massive amounts of raw data quickly** even on systems designed for personal-scale projects.

Because the Couchbase Java SDK supports RxJava and a SQL-like language called N1QL, there are many more options when it comes to querying the name data.

Choosing a great name with Apache Spark. What if the data set were in the hundreds of millions of documents? While RxJava might perform well, it is probably best to use something such as Spark because it was designed for massive amounts of data.

To use Spark to get a list of popular baby names based on a gender and a threshold, I might do something like this:

```
public void getPopularNames(
  String gender, int threshold) {
    String queryStr = "SELECT Name, Gender, " +
      "SUM(Count) AS Total FROM 'default' " +
      "WHERE Gender = $1 GROUP BY Name, " +
      "Gender HAVING SUM(Count) >= $2";
    JsonArray parameters = JsonArray.create()
       .add(gender)
       .add(threshold);
    ParameterizedN1qlQuery query =
      ParameterizedN1qlQuery.parameterized(
        queryStr, parameters);
    this.couchbaseSparkContext
        .couchbaseQuery(query)
        .foreach(queryResult ->
          System.out.println(queryResult));
}
```

Notice that the setup and parameterization is the same. The difference comes in where the query is actually executed. Instead of the built-in RxJava features of the Couchbase Java SDK, I use the Apache Spark Connector, which makes it possible to use Spark to run our application.

Conclusion

Let me be clear. I haven't actually chosen a name for my firstborn child, but if I wanted to make a decision in a statistical way through the use of RxJava and Apache Spark, I could.

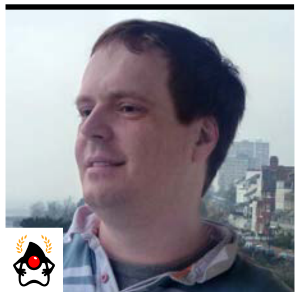
By using big data tools such as Apache Spark combined with a NoSQL database such as Couchbase, which offers a memory-centric architecture, you can process massive amounts of raw data quickly even on systems designed for personal-scale projects. </article>

Nic Raboy (@nraboy) is a developer advocate for Couchbase in the San Francisco Bay Area. He has released several native and hybrid mobile applications to iTunes and Google Play and writes about his development experiences with a focus on making web and mobile app development easier to understand.

learn more

[Code download from the *Java Magazine* download area]

[OpenCSV, a Java parser for CSV files]



PETER LAWREY

Building a Massive Off-Heap Data Queue

How one company built a data queue that scales to more than 100 GB

My company develops software for financial services firms, particularly for high-frequency trading. This kind of software requires the ability to store large amounts of transactional data in memory. By *large amounts*, I mean tens of gigabytes, sometimes more than 100 GB per day in real-time systems. For offline reporting, the largest data set we've imported into Java was 100 TB. In this article, I explain how we built a data structure—a specialized queue—that can manage terabytes of data off the heap. This article is intended for intermediate to advanced programmers.

The queue implementation, called Chronicle Queue, is an open source, persisted journal of messages. It supports concurrent writers and readers even across multiple JVMs on the same machine. Every reader sees every message, and a reader can join at any time and still see every message. In our applications, we assume that you can read and scan through messages fast enough that even if you aren't interested in most messages, getting at the information you want will still be fast enough.

In our design, readers are not consumers, so messages don't disappear after they're read. This message retention has multiple advantages when compared with the usual queue operation of message removal:

- A message can be replayed as many times as needed.
- A day of production messages can be replayed in testing months later.
- It reduces the requirement for logging almost entirely.

But, of course, it presents the problem of an ever-growing data set that needs to be managed in memory.

Designwise, the unbounded queue model removes the need for flow control between the writers and readers that is required with traditional queues, so that spikes in data inflow don't overflow the queue. There is no interaction between the writers and readers, and you can performance-test them in isolation and expect to get similar results when they're running at the same time.

Note: There can be a coordination overhead when multiple threads are writing to the same data structure. We have found this overhead to be significantly improved in Intel Haswell processors compared with Sandy Bridge processors. This coordination is implemented entirely using Java's atomic compare-and-set operations, without interacting with the operating system.

For developers, the retention of messages and absence of the need for flow control has these specific advantages:

- You can reproduce a bug even if it only occurs once in a million messages, by replaying all the messages that led to the bug triggering—but, more importantly, you can have confidence that *the bug*, rather than *a bug*, has been fixed.
- You can test a microservice replaying from the same input file repeatedly without the producer or downstream consumers running.

- You can test every microservice independently because there is no flow control interaction between them. If you have flow control between, let's say, 20 services, any one of those services could slow down any producer and, in turn, its producer until the entire system locks up. This is something you need to test for. Without flow control, this can't happen.

Inside Chronicle Queue

What might be surprising is that Chronicle Queue is written entirely in pure Java. It can outperform many data storage solutions written in C. You might be wondering how that's possible, given that well-written C is usually faster than Java.

One problem with any complex application is that you need a degree of protection between your services and your data storage to minimize the risk of corruption. As Java uses a JVM, it already has an abstraction layer and a degree of protection. If an application throws an exception, this doesn't mean the data structure is corrupted. To get a degree of isolation in C, many data storage solutions use TCP to communicate with large data structures. The overhead of using TCP (even over loopback) can exceed the performance benefit of using C. Because Chronicle Queue supports sharing of the data structure in memory for multiple JVMs, it eliminates the need to use TCP to share data.

Memory management. Chronicle Queue is built on a class called `MappedBytes` in the package `Chronicle-Bytes`. It implements the message queue as a memory-mapped file.

`MappedBytes` in turn maps the underlying file into memory in chunks, with an overlapping region to allow messages to easily pass from one chunk to another. The default chunk size is 64 MB, and the overlap is 16 MB. When you write just one byte, 80 MB (64 + 16) of virtual memory is allocated. When you get to the end of the first 64 MB, another region is mapped in and you get another 64 MB. It drops mappings on a least recently used (LRU) basis. The result is a region of memory that appears to be unbounded and can exceed the virtual memory limit of your machine, which is usually between 128 TB and 256 TB, depending on your operating system.

How do you load data into memory and save data? This is what the operating system does for you. It zeroes out memory pages you haven't used, loads from disk pages that have been used before, and saves data to disk asynchronously without the JVM being involved or even running. For example, if you write some data and your process dies, the data will still appear in cache and be written to disk. (That is, if the operating system didn't also die. To protect from operating system failure, we use TCP replication.)

Linux (and Windows similarly) allows up to 10 percent of main memory to be dirty/written to, but not saved to disk (with the default setting). A machine with 512 GB of main memory can have up to 51 GB of data uncommitted to disk. This is an enormous amount of data—and you can accept a burst of data this size with minimal impact on the application. Once this threshold is reached, the application is prevented from dirtying any new pages before some are written to disk.

Other Design Considerations

What if you don't have a big server? Even a PC with 8 GB will allow up to 800 MB of unwritten data. If your typical message size is 200 bytes, this is still a capacity for a sudden burst of 4 million messages. Even while these messages are being captured, data will be written to disk asynchronously. As writes are generally sequential, you can achieve the maximum write throughput of your disk subsystem.

Chronicle Queue also supports a memory-mapped HashMap. This is a good option if you need random access and your data set fits in memory; however, once your working set of data exceeds the main memory size, performance can drop by an order of magnitude.

A feature of the JVM that was essential for building this library was the use of `sun.misc.Unsafe`. While its use is highly discouraged, there wasn't a practical alternative up to

Java 8. In Java 9, I hope, we will see a replacement in the form of custom intrinsics, which promise to be a more powerful approach. What makes intrinsics so important for performance is that you don't pay the cost of using the Java Native Interface (JNI). While the cost is low, it's not low enough if you're calling it around a billion times per second. An intrinsic is a method that the JVM recognizes and replaces with machine code to do the same thing. Usually this is a native method, but it can even be a Java method.

By using a transparent format, we can validate the data structure and focus on portions of it at a time, allowing us to implement much more complex data structures.

The key benefit of using memory-mapped files is that you are no longer limited by the size of your heap, or even the size of your main memory. You are limited only by the amount of disk space you have. The cost of disk space is usually still much lower than main memory.

A surprising benefit of memory-mapped files is that you can use them in a mode in which there is only one copy in memory. This means that in the operating system's disk cache, the memory in process A and the memory in process B are shared. There is only one copy. If process A updates it, not only is this visible to process B in the time it takes the L2 caches to become coherent (that is, synchronize the data, which today typically takes around 25 nanoseconds), but the operating system can asynchronously write the data to disk. In addition, the operating system can predictively read ahead, loading data from disk when you access the memory (or files) sequentially.

Finally, an important benefit of using a memory-mapped file is the ability to bind a portion of memory to an object. Java doesn't have support for pointers to random areas of memory. We turn to an interface of getters, setters, and atomic operations and use off-heap memory as the storage and transport between processes. For example, the header of the Chronicle Queue has a field for the last acknowledged index so replication can be monitored. This is wrapped as a LongValue interface. When this object is read, written, or atomically updated, the off-heap memory it points to is accessed. This value is both shared between processes and persisted by the operating system without the need for a system call.

The data structure in detail. Each entry is a blob with a prefix of four bytes. The prefix contains one bit indicating whether the entry is user data or metadata needed to support the queue itself, another bit indicates whether the message in the entry is complete or not, and the remaining 30 bits contain the length of the message.

When the message is not complete, it cannot be read. But if the length is known, a writer can skip such messages and attempt to write after them. If Thread1 is in the middle of writing a message but it doesn't know how long it will be, it can write four bytes, which contains the length. Thread2 can see that there will be a message and skip over it looking for a place to write. This way multiple threads can write to the queue concurrently. Any message that is detected as bad, such as a thread that died while writing, can be marked as bad metadata and skipped by the reader.

As the files grow without limit, you may need to compress or delete portions while the data structure is still in use. To support this, we have time-based *file rolling*. By default you get one file per day. This allows you to manage data simply by compressing, copying, and deleting this daily file. The rolling can also be performed more or less often, as required.

There is a special value that is a "poison pill" value, which indicates that the file has been rolled. This ensures that all writers and readers roll to the new file at the same point in a timely manner.

For the message data itself, we use a binary form of YAML to store the messages because it's a format that is designed to

be human-readable. We view JSON and XML as technologies aligned with JavaScript and SGML, respectively, and not as readable. We support JSON for messaging with browsers. The binary form of YAML is used for performance reasons, and it can be automatically translated to YAML for viewing.

We didn't always use such a transparent format, and that placed a limit on how complex the data structure could be. By using a transparent format, we can validate the data structure and focus on portions of it at a time, allowing us to implement much more complex data structures as well as support backward compatibility for older formats.

Append-only benefits. Chronicle Queue is designed for sequential writes and reads. It also supports random access and updates in place, although you can't change the size of an existing entry. You can minimize this limitation by padding an entry for future use.

Sequential reads and writes are more efficient for persistence to HDD and SSD disks. The append-only structure makes replication much simpler as well.

Conclusion

Using native memory to complement managed, on-heap memory can allow your Java application to exploit all the memory of your server. It can even extend the available data structure to exceed main memory by flushing to disk.

By using a simple data lifecycle and a transparent data structure, you can move a significant portion of your data out of the heap. This can improve the performance, stability, and scalability of your application. </article>

Peter Lawrey (@PeterLawrey) has answered the most questions about Java, JVM, concurrency, memory, performance, and file I/O on StackOverflow.com (roughly 12,000), and he writes the Vanilla Java blog. He is the architect of Chronicle Software and a Java Champion.

JOSH JUNEAU

Big Data Best Practices for JDBC and JPA

Focus on the fundamentals so you're not overwhelmed by large amounts of data.

When it comes to handling big data in applications, there are several pitfalls to avoid. An application that runs perfectly fine on a small or medium-size database can fail once the database has increased in size. Failure points for applications working with large amounts of data can encompass a broad spectrum of areas, including memory, poor database transaction performance, and bad architecture. Whether your development team chooses to use JDBC or an object-relational mapping framework, the architecture of your application can mean the difference between success and failure.

In this article, I cover some best practices to be used for working with data via JDBC or the Java Persistence API (JPA), so that your application does not fail under the pressure of big data. I don't delve into any proprietary APIs or frameworks for working with big data or target any particular database options available via standard RDBMS or NoSQL. Rather, I provide basic strategies for configuring environments and tuning code, as well as best practices for working with large amounts of data via a Java application.

First Things First: Stored Procedures

Ask yourself this question: Why am I pulling this large amount of data into my application? If the answer is that you are trying to perform some calculations, analysis, or other processing on a very large result set, you might want to reconsider your technique. Most databases (specifically RDBMSs) contain a variety of built-in functions and procedures for performing processing of data directly within the database, whereas many NoSQL databases don't offer stored procedures. However, many NoSQL databases offer some type of function or stored code capability, although they're not often as capable as those offered by a standard database. Many database solutions also contain a language that enables you to develop procedures that can be executed directly within the database. For example, Oracle Database contains its own procedural language known as PL/SQL, which safely extends the SQL language. When working with big data, you can achieve huge performance gains by allowing the database, rather than the application, to perform analytical processing—unless, of course, there is an analytical processing requirement that can only be performed outside of the database.

The JDBC API and JPA both contain solutions for calling upon a database's stored procedures. It's also easy to pass or retrieve values with stored procedures, and the best part is that a single connection can be made to call upon the procedure, which can in turn process thousands of records. Listing 1 demonstrates how to call a database stored procedure using either JDBC or JPA.

Listing 1.

```
// Using JDBC to call upon a database stored
// procedure
```

```
    CallableStatement cs = null;

    try {
    cs = conn.prepareCall("{call DUMMY_PROC(?,?)}");
    cs.setString(1, "This is a test");
    cs.registerOutParameter(2, Types.VARCHAR);
    cs.executeQuery();

    // Do something with result
    String returnStr = cs.getString(2);

    } catch (SQLException ex){
        ex.printStackTrace();
    }

// Utilize JPA to call a database stored procedure
// Add @NamedStoredProcedureQuery to entity class
@NamedStoredProcedureQuery(
  name="createEmp", procedureName="CREATE_EMP",
  parameters = {
     @StoredProcedureParameter(
       mode= ParameterMode.IN,
       type=String.class,
       name="first"),
     @StoredProcedureParamter(
       mode = ParameterMode.IN,
       type=String.class,
       name="last")
})

// Calling upon stored procedure
StoredProcedureQuery qry =
   em.createStoredProcedureQuery("createEmp");
qry.setParameter("first", "JOSH");
qry.setParameter("last","JUNEAU");
qry.execute();
```

Many database solutions also support stored procedures written in Java. What better way to make the most of your Java programming skills than to put them to work inside the database? There are trade-offs, though, because working with a Java stored procedure still requires connection code to access data, albeit inside the database rather than in an application. Therefore, coding stored procedures in Java might prove to be more cumbersome than doing so with a language that is written for the database.

Sure, in the end any database-driven application will need to work directly with data in some manner. But be mindful of the difference between the work that can be performed in the database and that which truly needs to be performed within a Java application.

Getting the Configuration Correct

Typically, in their default configuration, application servers are configured for use with applications utilizing average I/O. Any configuration that might affect performance should be reviewed and configured appropriately before you attempt to use it on an application with large amounts of data.

Application-specific configurations. By default the JDBC driver will retrieve a fixed number of rows with each fetch. For example, the default number of rows fetched from Oracle Database is 10. That means if you are trying to return 1,000 rows, your application will need to perform 100 fetch operations to retrieve all rows. Now imagine if you had to multiply that by 1,000—that could produce quite a bottleneck.

> If you need to work with a specific set of data multiple times, **connections can be easily managed** by caching data.

The JDBC `setFetchSize()` can be used to provide a hint as to the number of rows that should be fetched. Listing 2 demonstrates how to specify the hint. If zero is specified, then the JDBC driver will ignore the value. As can be seen in the code, once the `Statement` has been created, simply set the desired fetch size. The `ResultSet` fetch size can also be

specified. This value overrides the `Statement` fetch size.

Listing 2.

```
String qry = "select …";
CreateConnection.loadProperties();
issuesList = new ArrayList();
try (Connection conn =
        CreateConnection.getConnection();
     Statement stmt = conn.createStatement();
     ResultSet rs = stmt.executeQuery(qry);) {
         stmt.setFetchSize(300);

         while (rs.next())
                . . .
     }
} catch (SQLException e) {
    // Log the exception
}
```

The same configuration can be specified utilizing JPA. Rather than explicitly setting the fetch size for JPA, you can provide a hint to "guide" the JDBC driver to fetch the desired number of rows to improve performance. The code in **Listing 3** demonstrates how to specify the fetch size for a JPA query when using the EclipseLink driver. The fetch size in this example is specified as a String value.

Listing 3.

```
public List<DukeIssues> findAllConfigureFetchSize(
        String fetchSize){
    Query qry = em.createQuery(
        "select object(o) from DukeIssues o");
    qry.setHint(
        "eclipselink.JDBC_FETCH_SIZE", fetchSize);
    return qry.getResultList();
}
```

It is imperative to test a big data application with different values to determine the value that provides the most benefit. Keep in mind that setting a larger fetch size will affect the amount of memory that an application requires. Be mindful when setting the fetch size, and also be sure to configure the amount of memory for your application server accordingly. Also be mindful of the other JDBC driver configurations that might come into play when working with large amounts of data.

Memory can also become a concern if you are caching a very large amount of data, so be sure to have your environment up to par.

Connection management. Connections can be very expensive, so a good practice when working with databases is to limit the number of connections that are required. This means that an application should make the most of each connection rather than being wasteful and performing only small amounts of work with each connection. If you need to work with a specific set of data multiple times, connections can be easily managed by caching data, where possible. If many inserts, updates, or deletes will be occurring, then you might need to perform transactions or bulk updates rather than opening a connection and disposing of it each time an update is made.

Connection pools that are managed by an application server play an important role in management of data, in general. Typically, there are default connection pool sizes put into place by an application server. Once all the connections are utilized, the application server will request more connections. Likewise, if a connection is not being utilized, it's placed back into the pool. Sizing a connection pool appropriately for the number of connections that an application will utilize is critical to good performance.

There are several ways to configure a connection pool. An application server connection pool can usually be configured via an administrative console, XML, or a command-line utility. For instance, in the case of GlassFish, you can modify

connection pool configurations using the administrative console or the `asadmin` command line utility. **Listing 4** demonstrates how to create a JDBC Connection Pool and JDBC Connection Pool Resource via the utility.

Listing 4.

```
asadmin create-jdbc-connection-pool \
--datasourceclassname \
oracle.jdbc.pool.OracleDataSource \
--restype javax.sql.DataSource \
--property user=dbuser:password=dbpassword:url=
   "jdbc:oracle\:thin\:@localhost\:1521\:MYDB" \
jdbc_conn-pool

asadmin create-jdbc-connection-pool \
--connectionpoolid myjdbc_oracle-pool jdbc/resource
```

[The `--property user=` line and the one after it should be entered as a single item. —*Ed.*]

To obtain connections from a pool, the `javax.sql.ConnectionPoolDataSource` interface can be utilized. The `ConnectionPoolDataSource` returns a `PooledConnection` object, which can then be utilized to obtain a `Connection`. Typically a `ConnectionPoolDataSource` is implemented by a Java Naming and Directory Interface connection object, and a `PooledConnection` object can be retrieved by invoking the `getConnection()` method, as shown in **Listing 5**. If you're using JPA, the `EntityManager` will handle obtaining connections, although there might need to be some configuration made within the Persistence Unit.

Listing 5.

```
ConnectionPoolDataSource cpds =
    (ConnectionPoolDataSource)
        initialCtx.lookup(jndiName);
PooledConnection pooledConn =
    ConnectionPoolDataSource.getConnection();
Connection conn = pooledConn.getConnection();
```

Another thing to keep in mind is the isolation level, which is how a database helps maintain data integrity. The lower the isolation level, the less uniform the system will become. Different isolation levels have their own use cases, and some also provide better performance than others. Study the impact of each isolation level on the application to determine the one that is best suited. **Table 1** lists the different isolation levels, from least consistent to most consistent, along with their performance impact.

A full overview of each isolation level is out of the scope of this article, but I mention this because it could be a factor in performance and data integrity.

Best Practices for Data Access Code

Once the configuration for an application and environment is correct, the query and statement code is the next place to look for signs of potential trouble. There are several best practices to follow when coding an application, and any code specifically dealing with data can have a major impact on the performance of an application if it deals with large amounts of data.

One of the first things to keep in mind is caching data. If an application will be utilizing a large number of records for read-only purposes, then it makes sense to cache the data so that it can be quickly accessed from memory rather than fetched from the database. Of course, memory can also

ISOLATION LEVEL	PERFORMANCE IMPACT
TRANSACTION_NONE	MAXIMUM SPEED
TRANSACTION_READ_UNCOMMITTED	MAXIMUM SPEED
TRANSACTION_READ_COMMITTED	FAST SPEED
TRANSACTION_REPEATABLE_READ	MEDIUM SPEED
TRANSACTION_SERIALIZABLE	SLOW SPEED

Table 1. Transaction isolation levels

become a concern if you are caching a very large amount of data, so be sure to have your environment up to par. Another coding practice that is important to get correct is utilizing the proper JDBC code for your application requirements. I will delve into some pointers to keep in mind when coding JDBC and JPA queries for large data access. Lastly, managing operations when performing updates and deletes can be detrimental to an application's success, and in this section I touch upon some techniques for performing bulk operations.

It is important to get the correct JDBC driver for your environment.

Caching data. When an application is required to work with the same data more than once, it makes sense to keep it in memory rather than incur additional database roundtrips. There are several ways to cache data that is frequently accessed into memory. One such solution is to utilize an in-memory data grid, such as Hazelcast. Another solution is to make use of built-in JDBC caching solutions, which often provide a more lean but less robust alternative to a data grid.

In-memory data grids make it easy to store data in distributed Maps, Lists, and Queues so that it can be utilized many times without making multiple trips to the database. This solution, in particular, is easy to get started with, yet it's advanced enough to provide an abundance of options for scaling, partitioning, and balancing your data. To make things nicer, Payara has built-in Hazelcast solutions.

If you are leaning toward bare-metal JDBC caching, utilization of JDBC solutions such as `javax.sql.CachedRowSet` make it possible to store smaller amounts of data for repeated access. Data within a `CachedRowSet` can be modified and later synchronized back to the database. A `CachedRowSet` can be generated from a `RowSetFactory` or a `CachedRowSet` default constructor. A `Connection` is then configured for the `CachedRowSet` object, and a String-based SQL command is passed containing the query. If the object will be used for making updates, then the primary keys of the table need to be specified. Lastly, the statement can be executed, returning the data. **Listing 6** demonstrates how to make use of a `CachedRowSet`.

Listing 6.

```
RowSetFactory factory;

try {
// Create RowSetFactory
    factory = RowSetProvider.newFactory();
// Create a CachedRowSet object using the factory
    crs = factory.createCachedRowSet();
// Populate the CachedRowSet connection settings,
//   if needed
// crs.setUsername(username);
// crs.setPassword(password);
// crs.setUrl(jdbc_url);
// Populate a query
    crs.setCommand("select id, request_date, "+
      "priority, description from duke_issues");
// Set key columns
    int[] keys = {1};
    crs.setKeyColumns(keys);
// Execute query
    crs.execute(conn);
// Manipulate the object contents in a
// disconnected state
    while (crs.next()) {
        // perform some work on the resultset
    }
} catch (SQLException ex) {
    ex.printStackTrace();
}
```

In JDBC, use PreparedStatements. First and foremost, if you are writing JDBC, use `PreparedStatements` rather than normal `Statements`. A `PreparedStatement` can be pre-

compiled, so if it is executed multiple times, it will not need to be recompiled each time. Not only will your application gain performance benefits but security benefits as well. `PreparedStatements` are advantageous for guarding against SQL injection. **Listing 7** demonstrates the typical use of a `PreparedStatement` for retrieving a `ResultSet`.

Listing 7.

```
public List<DukeIssues>
       queryIssues(String assignedTo) {
    String qry =
    "SELECT ID, REQUEST_DATE, PRIORITY, DESCRIPTION "
    + "FROM DUKE_ISSUES "
    + "WHERE assigned_to = ?";

    List<DukeIssues> issueList = new ArrayList();
    try (Connection conn =
             CreateConnection.getConnection();
         PreparedStatement stmt =
             conn.prepareStatement(qry))
    {
        stmt.setString(1, assignedTo);
        try (ResultSet rs = stmt.executeQuery();) {
          while (rs.next()) {
            int id = rs.getInt("ID");
            java.util.Date requestDate =
              rs.getDate("REQUEST_DATE");
            int priority = rs.getInt("PRIORITY");
            String description =
              rs.getString("DESCRIPTION");
            DukeIssues issue = new DukeIssues();
            issue.setId(id);
            issue.setRequestDate(requestDate);
            issue.setPriority(priority);
            issue.setDescription(description);
            issueList.add(issue);
          }
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
    return issueList;
}
```

It might be obvious, but make sure that `Statements` and `PreparedStatements` are closed once they've been used. Doing so gives the garbage collector an opportunity to recycle the memory.

Use bulk operations. If an application is performing many subsequent updates or deletes, then it might be best to perform these operations in bulk. Both JDBC and JPA provide the means for using bulk write and delete operations when needed. To fully understand if bulk operations will be helpful to your application, it's important to understand how they work. There are a couple of different types of batch writing: *parameterized* and *dynamic*.

Parameterized batch writing essentially takes a bulk number of identical inserts, updates, or deletes and chains them together, using bind variables for parameters. In other words, the only thing that changes throughout the bulk of the operation is the parameters—the SQL for the operation remains the same. The chain of operations is then sent to the database in a single call and executed in bulk. Parameterized batch writing improves performance twofold because the same statement is utilized for each operation, so the SQL doesn't need to be parsed each time, and only one network connection is used because everything is sent in bulk.

Data management is the first step toward writing an application to work with large amounts of data.

Dynamic batch writing allows each SQL statement in a bulk operation to contain different SQL, so more than one heterogeneous statement can be sent in bulk to the database. Parameter binding is

not allowed with dynamic batch writing, so the SQL must be parsed for each statement. Therefore, the net gain on performance is only database connection-related, and it might not be as beneficial as that of a parameterized batch operation.

Determining which type of bulk operation is being used by your JDBC driver involves some amount of testing and investigation. JDBC contains a standard API that allows you to determine which type of bulk operation you can perform. To utilize batch operations with raw JDBC, first set auto-commit on the connection to false. By default, each operation is committed automatically, and by setting auto-commit to false the operations will be executed but not committed until an explicit commit is issued. Listing 8 shows a simple example of how to perform a group of inserts and updates in a batch operation. In a big data environment where many rows are being inserted at a time, the insert statements may be issued in a looping construct: first opening the connection, next looping through the inserts, and finally committing at the end and closing the connection.

Listing 8.

```
List<DukeIssues> issueList = queryIssues("JUNEAU");

String insStmt1 =
"insert into duke_issues (id, request_date," +
"priority, description) values " +
"(908472, '2016-01-01',0,'QUERY NOT WORKING')";

String insStmt2 = "insert into duke_issues " +
"(id, request_date, priority, description) values "
+
"(908473, '2016-01-01',0,'RESULTS NOT POSTING')";

String updStmt = "insert duke_issues " +
"set status = ? where assigned_to = ?";

try (Connection conn = getConnection();
        Statement stmt = conn.createStatement();) {
    conn.setAutoCommit(false);

    // Perform loop here to add statements to the
    // batch transaction, if needed.

    stmt.addBatch(insStmt1);
    stmt.addBatch(insStmt2);
    stmt.addBatch(updStmt);
    int[] count = stmt.executeBatch();

    conn.commit();
    conn.setAutoCommit(true);
} catch (SQLException e) {
    // Log the exception
}
```

There is no prescribed standard for performing batch operations using JPA. However, most of the JPA providers do support some type of batching. To enable batch operations to occur within a JPA environment, typically configurations must be made within the persistence unit. Most JPA drivers accommodate parameterized and dynamic batch writing, so you must configure accordingly. Listing 9 demonstrates how to configure for EclipseLink within the persistence unit.

Listing 9.

```
<persistence-unit>
. . .
<property name="eclipselink.jdbc.batch-writing"
     value="JDBC"/>
<property name="eclipselink.jdbc.batch-writing.size"
     value="1000"/>
. . .
</persistence-unit>
```

Consider a specialized driver or API. It is important to get the correct JDBC driver for your environment. Although the database vendor may provide a JDBC driver, it might not be the most optimal driver for your application's use case. There are

many drivers that have been specifically tuned for working with large amounts of data. Make sure to choose one of those. In other words, do your homework with respect to the database platform and the environment you'll be working in to ensure you have the best driver for the job.

Conclusion

Before digging into the nuts and bolts of specific APIs, data management is the first step toward writing an application to work with large amounts of data. Data management strategies include configuring JDBC and JPA environments accordingly, coding for the best performance, and caching as needed. Get the fundamentals of the application working correctly to avoid developing an application that doesn't scale or perform well. In this article, I looked at the basics. But once those items are in place, it's important that JDBC and JPA environments constantly be monitored for performance to get the best possible performance for a given data load. `</article>`

Josh Juneau (@javajuneau) works as an application developer, system analyst, and database administrator. He is a technical writer for Oracle Technology Network and *Java Magazine*. He has written books on Java and Java EE for Apress and is also a JCP Expert Group member for JSR 372 and JSR 378.

learn more

[Oracle's JDBC tutorial]

[Oracle's JPA tutorial]

[Background on database isolation]

BARCELONA JUG

Barcelona, Spain, has a vibrant startup ecosystem and many business communities related to software. It's a beautiful city with lots of places to visit and some of the best urban beaches in the world. It's also home to the Barcelona Java Users Group (@barcelonajug), a non-profit association built around a team with broad experience in Java technologies. Since 2012, it has been organizing talks and other get-togethers focused on Java topics, usually once a month. Most of the meetings are held in English.

Some past topics include developer tools, testing techniques, API design, and high-performance messaging. The group has hosted talks from Stephen Chin (Oracle); Claus Ibsen, Mario Fusco, Gavin King, and Mauricio Salatino (Red Hat); Jean-Baptiste Onofré (Talend); Alex Soto (CloudBees); Peter Kriens (OSGi Alliance); and Norberto Leite (MongoDB), among others.

Its big event last year was the Java Barcelona Conference, a two-day event focused on Java, JVM, and related technologies, as well as open source technologies. Developers from several countries came to learn and explore software development in a unique environment. The talks given at the 2015 event can be viewed online.

The group is organizing this year's event, which will take place June 16 to 18, with even more topics and networking opportunities. This year, the event will take place at Pompeu Fabra University and include hands-on workshops. You can learn more and buy tickets here.

For more about the Barcelona JUG, see Meetup or YouTube.

MERT ÇALIŞKAN

JUnit 5: A First Look

The long-awaited release of JUnit 5 is a complete redesign with many useful additions.

JUnit, the widely used Java unit testing framework, has just seen the first alpha release of version 5 after 10 years on version 4. JUnit 5 consists of a revamped codebase with a modular architecture, a new set of annotations, an extensible model for third-party library integration, and the ability to use lambda expressions in assertions. The predecessor of JUnit 5 was the JUnit Lambda project, which sowed the first ideas for the next generation of unit testing and was crowd-funded until October 2015.

Through the years, JUnit has captured the essence of what a unit testing framework should be. However, its core mostly stayed intact, which made it difficult for it to evolve. This new version is a complete rewrite of the whole product that aims to provide a sufficient and stable API for running and reporting tests. Implementing unit tests with JUnit 5 requires Java 8 at a minimum, but it can run tests on code written for earlier versions of Java.

In this article, I describe the principal features of this early release of JUnit 5, illustrating them with detailed examples. All the code in this article is based on JUnit version 5.0.0-ALPHA, which was released in February 2016. The complete source code for the examples in this article is available at the *Java Magazine* download area.

The JUnit team is planning to ship a release candidate of the framework in the third quarter of 2016. Milestone 1 is one of the last steps before JUnit 5 officially ships. This will surely be one of the most consequential releases ever in the Java ecosystem.

Configuring Tools to Use JUnit 5

JUnit 5 dependency definitions are available for both Maven and Gradle. For this article, I used Maven and its dependency definition for the JUnit 5 API. The following shows the Maven inclusion of JUnit 5:

```
<dependency>
    <groupId>org.junit</groupId>
    <artifactId>junit5-api</artifactId>
    <version>5.0.0-ALPHA</version>
    <scope>test</scope>
</dependency>
```

JUnit 5 now consists of multiple modules including the `junit5-api` module, which provides a transitive dependency, and `opentest4j`, which is named after the Open Test Alliance for the JVM project. Its aim is to provide a minimal common foundation for testing libraries, IDEs, and other tools such as TestNG, Hamcrest, Spock, Eclipse, Gradle, Maven, and many others.

In addition, JUnit 5 has the following modules:

- `junit5-api`, an API module that contains classes for implementing tests.
- `junit4-engine`, a JUnit 4 engine implementation. It locates and runs JUnit 4–based tests.
- `junit5-engine`, a JUnit 5 engine implementation module. It locates and runs JUnit 5–based tests.
- `junit-engine-api`, an abstraction API module for testing engines. It provides an extensible mechanism with

which current and feature testing frameworks can integrate themselves by registering their test engines. Test engines are identified by an ID string, and the engines are located via reflection through the class loader. Test engines register themselves via the JDK's `ServiceLoader` class.

- `junit-launcher`, an integration module that is used by build tools and IDEs to run tests. It makes it possible to run JUnit 4 and JUnit 5 tests in a single run.
- `junit-console`, an API module for running JUnit 4 and JUnit 5 tests from the command line. It prints execution results to the console.
- `junit-commons`, a common API module that is being used by all modules.
- `junit4-runner`, an API module for running JUnit 5 tests on JUnit 4. This eases the migration of JUnit 4–based implementations to JUnit 5, because the IDEs and the build tools don't support JUnit 5 tests yet.
- `surefire-junit5`, a module that contains `JUnitGen5Provider`, which integrates with the Maven Surefire plugin for running JUnit 5 tests on JUnit 4.
- `junit-gradle`, a module that contains `JUnit5Plugin`, which integrates with Gradle builds for running JUnit 5 tests on JUnit 4.

One of the main goals of JUnit 5 modules is to decouple the API for executing the tests from the APIs for implementing the tests.

Anatomy of a JUnit 5 Test

Let's look at some JUnit 5 tests, starting with the simple JUnit test shown in **Listing 1**.

Listing 1.

```
import org.junit.gen5.api.Test;

class SimpleTest {

    @Test
    void simpleTestIsPassing() {
        org.junit.gen5.api.Assertions.
    assertTrue(true);
    }
}
```

For a simple JUnit 5 test class, such as the one shown in **Listing 1**, there is almost no difference to be seen at first glance when compared with a JUnit 4 test class. The main difference is that there is no need to have test classes and methods defined with the `public` modifier. Also, the `@Test` annotation—along with the rest of the annotations—has moved to a new package named `org.junit.gen5.api`, which must be imported.

Capitalizing on the Power of Annotations

JUnit 5 offers a revised set of annotations, which, in my view, provide essential features for implementing tests. The annotations can be declared individually or they can be composed to create custom annotations. In the following section, I describe each annotation and give details with examples.

@DisplayName. It's now possible to display a name for a test class or its methods by using the `@DisplayName` annotation. As shown in **Listing 2**, the description can contain spaces and special characters. It can even contain emojis such as ☺.

Listing 2.

```
@DisplayName("This is my awesome test class")
class SimpleNamedTest {

    @DisplayName("This is my lonely test method")
    @Test
    void simpleTestIsPassing() {
        assertTrue(true);
    }
}
```

@Disabled. The `@Disabled` annotation is analogous to the `@Ignore` annotation of JUnit 4, and it can be used to disable the whole test class or one of its methods from execution. The reason for disabling the test can be added as description to the annotation, as shown in Listing 3.

Listing 3.

```
class DisabledTest {

    @Test
    @Disabled("test is skipped")
    void skippedTest() {
        fail("feature not implemented yet");
    }
}
```

@Tags and @Tag. It's possible to tag test classes, their methods, or both. Tagging provides a way of filtering tests for execution. This approach is analogous to JUnit 4's `Categories`. Listing 4 shows a sample test class that uses tags.

Listing 4.

```
@Tag("marvelous-test")
@Tags({@Tag("fantastic-test"),
       @Tag("awesome-test")})
class TagTest {

    @Test
    void normalTest() {
    }

    @Test
    @Tag("fast-test")
    void fastTest() {
    }
}
```

You can filter tests for execution or exclusion by providing tag names to the test runners. The way of running `ConsoleRunner` is described in detail shortly. With `ConsoleRunner`, you can use the `-t` parameter for providing required tag names or the `-T` parameter for excluding tag names.

@BeforeAll, @BeforeEach, @AfterEach, and @AfterAll. The behavior of these annotations is exactly the same as the behavior of JUnit 4's `@BeforeClass`, `@Before`, `@After`, and `@AfterClass`, respectively. The method annotated with `@BeforeEach` will be executed before each `@Test` method, and the method annotated with `@AfterEach` will be executed after each `@Test` method. The methods annotated with `@BeforeAll` and `@AfterAll` will be executed before and after the execution of all `@Test` methods. These four annotations are applied to the `@Test` methods of the class in which they reside and they will also be applied to the class hierarchy, if any exists. (See the next section on test hierarchies.) The methods annotated with `@BeforeAll` and `@AfterAll` need to be defined as static.

@Nested test hierarchies. JUnit 5 supports creating hierarchies of test classes by nesting them inside each other. This option enables you to group tests logically and have them under the same parent, which facilitates applying the same initialization methods for each test. Listing 5 shows an example.

Listing 5.

```
class NestedTest {

    private Queue<String> items;

    @BeforeEach
    void setup() {
        items = new LinkedList<>();
    }

    @Test
    void isEmpty() {
        assertTrue(items.isEmpty());
    }
```

```
    @Nested
    class WhenEmpty {
      @Test
      public void removeShouldThrowException() {
        expectThrows(
          NoSuchElementException.class,
          items::remove);
        }
      }

    @Nested
    class WhenWithOneElement {
      @Test
      void addingOneElementShouldIncreaseSize() {
          items.add("Item");
          assertEquals(items.size(), 1);
      }
    }
}
```

Assertions and Assumptions

The `org.junit.gen5.Assertions` class of JUnit 5 contains static assertion methods—such as `assertEquals`, `assertTrue`, `assertNull`, and `assertSame`—and their corresponding negative versions for handling the conditions in test methods. JUnit 5 leverages the use of lambda expressions with these assertion methods by providing overloaded versions that take an instance of `java.util.function.Supplier`. This enables the evaluation of the assertion *message* lazily, meaning that potentially complex calculations are delayed until a failed assertion. **Listing 6** shows using a lambda expression in an assertion.

Listing 6.

```
class AssertionsTest {

    @Test
    void assertionShouldBeTrue() {
        assertEquals(2 == 2, true);
    }

    @Test
    void assertionShouldBeTrueWithLambda() {
        assertEquals(3 == 2, true,
              () -> "3 not equals to 2!");
    }
}
```

The `org.junit.gen5.Assumptions` class provides `assumeTrue`, `assumeFalse`, and `assumingThat` static methods. As stated in the documentation, these methods are useful for stating assumptions about the conditions in which a test is meaningful. If an assumption fails, it does not mean the code is broken, but only that the test provides no useful information. The default JUnit runner ignores such failing tests. This approach enables other tests in the series to be executed.

Grouping Assertions

It's also possible to group a list of assertions together. Using the `assertAll` static method, which is shown in **Listing 7**, causes all assertions to be executed together and all failures to be reported together.

Listing 7.

```
class GroupedAssertionsTest {

    @Test
    void groupedAssertionsAreValid() {
        assertAll(
            () -> assertTrue(true),
            () -> assertFalse(false)
        );
    }
}
```

Expecting the Unexpected

JUnit 4 provides a way for handling exceptions by declaring them as an attribute to the `@Test` annotation. This is an enhancement compared with previous versions that required the use of `try-catch` blocks for handling exceptions. JUnit 5 introduces the usage of lambda expressions for defining the exception inside the assertion statement. Listing 8 shows the placement of the exception directly into the assertion.

Listing 8.

```
class ExceptionsTest {

    @Test
    void expectingArithmeticException() {
        assertThrows(ArithmeticException.class,
                     () -> divideByZero());
    }

    int divideByZero() {
        return 3/0;
    }
}
```

With JUnit 5, it's also possible to assign the exception to a variable in order to assert conditions on its values, as shown in Listing 9.

Listing 9.

```
class Exceptions2Test {

  @Test
  void expectingArithmeticException() {
    StringIndexOutOfBoundsException exception =
      expectThrows(
        StringIndexOutOfBoundsException.class,
        () -> "JUnit5 Rocks!".substring(-1));

    assertEquals(exception.getMessage(),
        "String index out of range: -1");
  }
}
```

Parameterized Test Methods

With JUnit 5, it's now possible to have test methods with parameters for the default runner implementation. This option enables dynamically resolved parameters to be injected at the method level. `TestInfoParameterResolver` and `TestReporterParameterResolver` are two built-in resolvers shipping with JUnit 5 that achieve this.

If the method parameter is an instance of `org.junit.gen5.api.TestInfo`, then `TestInfoParameterResolver` will supply an instance of it as a parameter. The test name or its display name can be retrieved from a `TestInfo` instance.

If the method parameter is an instance of `org.junit.gen5.api.TestReporter`, then `TestReporterParameterResolver` will supply an instance of it as a parameter. Key-value pairs could be published to the reporter instance in order to be used by IDEs or reporting tools, as in the example shown in Listing 10.

Listing 10.

```
class ResolversTest {

  @Test
  @DisplayName("my awesome test")
  void shouldAssertTrue(
         TestInfo info, TestReporter reporter)
      {
         System.out.println(
           "Test " + info.getDisplayName() +
           " is executed.");

         assertTrue(true);
         reporter.publishEntry(
           "a key", "a value");
```

```
    }
}
```

Instances of `TestInfo` or `TestReporter` can also be injected into methods annotated with `@BeforeAll`, `@BeforeEach`, `@AfterEach`, and `@AfterAll`.

The New Extension Model

JUnit 5 provides an extensible API for combining various extension points. It gives new options to third parties for integrating their own frameworks with the testing mechanism of JUnit. With JUnit 4, this was possible with `Runner`, `@Rule`, and `@ClassRule`, but JUnit 5 offers a unified API with which multiple extensions can be defined with `@ExtendWith`. Listing 11 is an example that employs Mockito framework integration.

Listing 11.

```
public interface User {
    String getName();
}

@ExtendWith(MockitoExtension.class)
class ExtensionsTest {

    @BeforeEach
    void init(@InjectMock User user) {
        when(user.getName()).thenReturn("Mert");
    }

    @Test
    void parameterInjectionWorksOk(
           @InjectMock User user) {
        assertEquals("Mert", user.getName());
    }
}
```

The `MockitoExtension` class is an implementation of `MethodParameterResolver`, which is an extension point for dynamically resolving method parameters at runtime. So the parameter annotated with the marker annotation `@InjectMock` will be mocked and stored through methods annotated with `@BeforeEach` and `@Test`.

You can access the source code for `MockitoExtension` and `@InjectMock` online.

Running JUnit 5

Currently, there is no support for directly running JUnit 5–based tests in IDEs and build tools, although there soon will be. JUnit 5 offers two ways for integrating and running JUnit 5–based tests. One is the `ConsoleRunner`, which is a command-line application for running tests, and the other is the JUnit 4 runner, which handles running tests on JUnit 4–integrated IDEs and build tools. The JUnit 4 runner provides integration with both Gradle and Maven build systems.

The `ConsoleRunner` can be executed with the Java command shown below. Building the classpath with the needed JAR files is a prerequisite for having the `ConsoleRunner` execute successfully, so ensure that you have correct version of the artifacts.

```
java -cp
  /path/to/junit-console-5.0.0-ALPHA.jar:
  /path/to/jopt-simple-4.9.jar:
  /path/to/junit-commons-5.0.0-ALPHA.jar:
  /path/to/junit-launcher-5.0.0-ALPHA.jar:
  /path/to/junit-engine-api-5.0.0-ALPHA.jar
   org.junit.gen5.console.ConsoleRunner
```

[Note that the command above should be entered as a single command. —*Ed.*]

It's also possible to run JUnit 5 tests in an IDE that supports only JUnit 4. To enable this, the `org.junit.gen5.junit4.runner.JUnit5` class should be defined with the `@RunWith` annotation, as shown in Listing 12.

Listing 12.

```
@RunWith(JUnit5.class)
public class RunWithTest {

    @Test
    public void simpleTestIsPassing() {
      org.junit.gen5.api.Assertions.
    assertTrue(true);
    }
}
```

The `junit4-runner` and `junit5-engine` module dependency should be defined in the classpath along with the JUnit 4 dependency.

Conclusion

The JUnit team has succeeded in offering a new, redesigned version of JUnit that addresses nearly all the limitations of previous versions. Note that the JUnit 5 API is still subject to change; the team is annotating the public types with the `@API` annotation and assigning values such as `Experimental`, `Maintained`, and `Stable`.

Give JUnit 5 a spin, and be prepared for the release that'll hit the streets in late 2016. Keep your green bar always on! </article>

Mert Çalişkan (@mertcal) is a Java Champion and coauthor of *PrimeFaces Cookbook* (first edition, Packt Publishing, 2013; second edition, 2015) and *Beginning Spring* (Wiley Publications, 2015). He is the founder of AnkaraJUG, which is the most active Java user group in Turkey.

learn more

[JUnit 5 official documentation]

Understanding Generics

Use generics to increase type safety and readability.

MICHAEL KÖLLING

In this installment of the "New to Java" series, I want to talk about *generics*.

If you have programmed for a little while in Java, it is likely that you have come across generics, and you have probably used them. (They are hard to avoid when using collections, and it is hard to do anything really interesting without collections.) If you are coming to Java from C++, you might have encountered the same concept as generics under the name of *parameterized types* or *templates*. (Templates in C++ are not the same as generics in all aspects, but they are closely related.)

Many novice Java programmers use generics without a full understanding of how they work and what they can do. This gap is what I address in this article.

In Java, the concept of generics is simple and straightforward in principle but tricky in the details. There is much to say about the corner cases, and it is also interesting to look into how generics are implemented in the Java compiler and the JVM. This knowledge helps you understand and anticipate some of the more surprising behaviors.

My discussion is spread over two parts. In this issue, I discuss the principles and fundamental ideas of generic types. I look at the definition and use of generics and provide a basic, overall understanding. In the next issue of *Java Magazine*, I will look at the more subtle parts, advanced uses, and implementation. If you read both articles, you will arrive at a good understanding of how generics can help you write better code.

PHOTOGRAPH BY JOHN BLYTHE

A Bit of History

Before Java 5 was released in 2004, Java did not have generic types. Instead, it had what are called *heterogeneous collections*. Here is what an example of a heterogeneous list looked like in those days:

```
List students = new ArrayList();
```

(Knowing this history is important, because you can still write this code in Java today—even though you shouldn't. And you might come across these collections in legacy code you maintain.)

In the example above, I intend to create a list of `Student` objects. I am using subtyping—the declared type of the variable is the interface `List`, a supertype of `ArrayList`, which is the actual object type created. This approach is a good idea because it increases flexibility. I can then add my student objects to the list:

```
students.add(new Student("Fred"));
```

When the time comes to get the student object out of my list again, the most natural thing to write would be this:

```
Student s = students.get(0);
```

This, however, does not work. In Java, in order to give the `List` type the ability to hold elements of any type, the `add` method was defined to take a parameter of type `Object`, and

the `get` method equally returns an `Object` type. This makes the previous statement an attempt to assign an expression of type `Object` to a variable of type `Student`—which is an error.

In the early days of Java, this was fixed with a cast:

```
Student s = (Student) students.get(0);
```

Now, writing this cast was always a mild annoyance. You usually know what types of objects are stored in any particular list—why can't the compiler keep track of them as well?

But the problem goes deeper than mere annoyance. Because the element type of the list is declared to be `Object`, you can actually add objects of any type to the same list:

```
students.add(new Integer(42));
students.add("a string");
```

The fact that this is possible—that the same list can hold objects of different types—is the reason it is referred to as *heterogeneous*: a list can contain mixed element types.

Having lists of different, unrelated element types is rarely useful, but it is easily done in error. The problem with heterogeneous lists is that this error cannot be detected until runtime. Nothing prevents you from accidentally inserting the wrong element type into the student list. Worse, even if you get the element out of the list and cast it to a `Student`, the code compiles. The error surfaces only at runtime, when the cast fails. Then a runtime type error is generated.

The problem with this runtime error is not only that it occurs late (at runtime, when the application might already have been delivered to a customer), but also that the source location of the detected error might be far removed from the actual mistake: you are notified about the problem when getting the element out, while the actual error was made when putting the element in. This might well be in a different part of the program entirely.

> It is useful to understand one aspect that changed slightly when generics entered the Java language: **the relationship between classes and types.**

Java and Type Safety

Java was always intended to be a type-safe language. This means that type errors should be detected at compile time, and runtime type errors should not be possible. This aim was never achieved completely, but it's a goal that the language designers strove for as much as possible. Casting breaks this goal: every time you use a cast, you punch a hole in type safety. You tell the type checker to look the other way and just trust you. But there is no guarantee that you will get things right.

Many times, when a cast is used, the code can be rewritten: often better object-oriented techniques can be used to avoid casting and maintain type safety. However, collections presented a different problem. There was no way to use them without casting, and this jarred with the philosophy of Java. That such an important area of programming could not be used in a type-safe way was a real annoyance. Thus in 2004, the Java language team fixed this problem by adding generics to the Java language.

Type Loss

The term for the problem with heterogeneous collections is *type loss*. In order to construct collections of any type, the element type of all collections was defined to be `Object`. The `add` method, for example, might be defined as follows:

```
public void add(Object element)
```

This works very well to put elements—say, objects of type Student—into the collection, but it creates a problem when you want to get them out again. Even though the runtime system will know that the element is of type Student, the compiler does not keep track of this. When you later use a get method to retrieve the element, all the compiler knows is that the element type is Object. The compiler loses track of the actual element type—thus the term *type loss*.

Introduction to Generics

The solution to avoid type loss was to give the compiler enough information about the element type. This was done by adding a type parameter to the class or interface definition. Consider an (incomplete and simplified) definition of an ArrayList. Before generics, it might have looked like this:

```
class ArrayList {
    public void add(Object element);
    public Object get(int index);
}
```

The element type here is Object. Now, with the generic parameter, the definition looks as follows:

```
class ArrayList<E> {
    public void add(E element);
    public E get(int index);
}
```

The E in the angle brackets is a type parameter: here, you can specify what the element type of the list should be. You no longer create an ArrayList object for the Student elements by writing this:

```
new ArrayList()
```

Instead, you now write this:

```
new ArrayList<Student>()
```

Just as with parameters for methods, you have a formal parameter specification in the definition (the E) and an actual parameter at the point of use (Student). Unlike method parameters, the actual parameter is not a value but a type.

By creating an ArrayList<Student> (which is usually read out loud as "an ArrayList of Student"), the other mentions of the type parameter E in the specification are also replaced with the actual type parameter Student. Thus, the parameter type of the add method and the return type of the get method are now both Student. This is very useful: now only Student objects can be added as elements, and you retrieve Student objects when you get them out again—no casting is needed.

When generics were introduced, a useful shortcut notation—the diamond notation— was provided to ensure that the increased readability does not lead to unnecessary verboseness.

Abstraction over Types

It is useful to understand one aspect that changed slightly when generics entered the Java language: the relationship between classes and types. Prior to generics, each class defined a type. For example, if you define a class Hexagon, then you automatically get a type called Hexagon to use in variable and parameter definitions. There is a very simple one-to-one relationship.

With generic classes, this is different. A generic class does not define a type—it defines a set of types. For example, the class ArrayList<E> defines the types ArrayList<Student>, ArrayList<Integer>, ArrayList<String>, ArrayList<ArrayList<String>>,

and any other type that can be specified by replacing the type parameter E with a concrete type.

In other words, generics introduce an abstraction over types—a powerful new feature.

The Benefits

One benefit of using generic classes should now be obvious: improved correctness. Incorrect types of objects can no longer be entered into a list. While erroneous attempts to insert an element could previously be detected only during testing (and testing can never be complete), they are now detected at compile time, and type correctness is guaranteed. In addition, if there is such an error, it will be reported at the point of the incorrect insertion—not at the point of retrieving the element, which is far removed from the actual error location.

The use of generics can make code safer and easier to read.

There is, however, a second benefit: readability. By explicitly specifying the element type of collections, you are providing useful information to human readers of your program as well. Explicitly saying what type of element a collection is intended for can make life easier for a maintenance programmer.

The Diamond Notation

When generics were introduced, a useful shortcut notation—the *diamond notation*—was provided to ensure that the increased readability does not lead to unnecessary verboseness.

Consider the very common case of declaring a variable and initializing it with a newly created object:

```
ArrayList<String> myList =
    new ArrayList<String>();
```

In some generic types, especially when there is more than one generic parameter, this line can get rather long:

```
HashSet<Integer, String> mySet =
    new HashSet<Integer, String>();
```

And it can get worse if a type parameter itself is generic:

```
HashSet<Integer, ArrayList<String>> mySet =
    new HashSet<Integer, ArrayList<String>>();
```

In each of these examples, the same lengthy generic type is spelled out twice: once on the left for the variable declaration and once on the right for the object creation. In this situation, the Java compiler allows you to omit part of the second mention of the type and instead write this:

```
HashSet<Integer, String> mySet = new HashSet<>();
```

Here, the generic parameters are omitted from the right side (leaving the angle brackets to form a diamond shape, thus the term *diamond notation*). This is allowed in this situation and means that the generic parameters on the right are the same as those on the left. It just saves some typing and makes the line easier to read. The semantics are exactly the same as they would be had you written out the types in full.

Summary—So Far

This was the easy part. The use of generics can make code safer and easier to read. Writing a simple generic class is quite straightforward, and creating objects of generic types is as well. You should also be able to read the documentation of simple generic types, such as the `List` interface's Javadoc page or the Javadoc page of the `Collection` interface.

However, the story does not end here. So far, I have ignored some problems that arise with generics, and understanding the mechanisms to solve them gets a little trickier. This

is where things become really interesting. Let's look at the problem first.

Generics and Subtyping

Assume that you have a small inheritance hierarchy. To model people in a university, you have classes `Student` and `Faculty`, and a common superclass `Person` (`Student` and `Faculty` are both subclasses of `Person`). So far, so good.

Now you also create types for lists of each of these: `List<Student>`, `List<Faculty>`, and `List<Person>`.

The student and faculty lists are held in the parts of your program that hold and process the student and faculty objects. The `Person` list type can be useful as a formal parameter for a method that you want to use with both faculty and students, for example:

```
private void printList(List<Person> list)
```

The idea is that you want to be able to call `printList` with both the student and faculty lists as actual parameters. This will work if the types of these lists are subtypes of `List<Person>`. But are they?

In other words, if `Student` is a subtype of `Person`, is then `List<Student>` a subtype of `List<Person>`?

Intuitively, you might say yes. Unfortunately, the correct answer is no.

You can see the problem when you imagine that the `printList` method not only prints, but also modifies the list passed as a parameter. Assume that this method inserts an object of type `Faculty` into the list. (Because the list is declared in the parameter as `List<Person>`, and `Faculty` is a subtype of `Person`, this is perfectly legal.) However, the actual list passed in to this method might have been a `List<Student>`. Then, suddenly, a `Faculty` object has been inserted into the student list! This is clearly a problem.

The only way to avoid this problem is to avoid considering lists of subtypes and lists of supertypes to be in a subtype/supertype relationship themselves. In other words, `List<Student>` is not a subtype of `List<Person>`.

Conclusion

There are many situations in which you need subtyping with generic types, such as the above attempt to define the generalized `printList` method. You have seen that it does not work with the constructs I have discussed so far, but just saying it can't be done is not good enough—you do need to be able to write such code.

The solutions entail additional constructs for generics: *bounded types* and *wildcards*. These concepts are powerful, but have some rather tricky corner cases. I will discuss them in the upcoming installment in the next issue of *Java Magazine*.

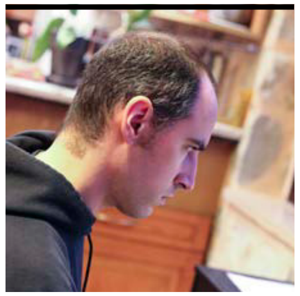
Until then, study the generic classes available in the Java library—especially the collections—and get used to the notation discussed in this article. I will dive deeper next time! </article>

Michael Kölling is a Java Champion and a professor at the University of Kent, England. He has published two Java textbooks and numerous papers on object orientation and computing education topics, and he is the lead developer of BlueJ and Greenfoot, two educational programming environments. Kölling is also a Distinguished Educator of the ACM.

learn more

[Oracle's Java tutorial on generics]()

[Wikipedia article on Java generics]()



STÉPHANE ÉPARDAUD

Ceylon Language: Say More, More Clearly

A low-ceremony, high-productivity JVM language that integrates easily with Java and also runs on JavaScript VMs

Ceylon is a modern statically typed JVM language, designed for readability and maintainability, while at the same time being expressive enough to allow efficient constructs with very little boilerplate. People familiar with Java or C# will have an easy time getting used to Ceylon because its syntax is familiar.

When I joined the project (led by Gavin King and Red Hat) almost five years ago, I was blown away by its ideas and goals. Finally, here was a new language made with the same philosophy that I loved in Java when it was created: a new language made to be familiar, yet more powerful, more expressive, with higher-level abstractions, leading to less and clearer code. But on top of that Ceylon also promised great tooling, a brand-new modern SDK, and features (at the time) missing from my other favorite language, Java—modularity, first-class functions, and reified generics. And it promised to remove lots of other frustrations.

Since that time, thanks to the many contributors who joined the project, the Ceylon team has delivered on its promises. Its first production-ready release was completed two years ago, with two new releases since then, and improvements and new features keep coming steadily.

It is impossible to cover the entire language in this article, so I highlight only a few important features to give you a sense of the language.

About Modularity

Ceylon has featured a modular architecture from the start. Every Ceylon package belongs to a module. If you declare your module, you can import other modules. If you do not, your code will be part of the default module. This code is what you need to write to start a trivial "Hello World" web application:

```
module hello "1" {
   import ceylon.net "1.2.2";
}

shared void run() {
   // create the server
   value server = newServer {
       Endpoint {
         path = startsWith("/");
         service(Request request, Response response)
             => response.writeString("hello world");
       }
   };

   // start it on port 8080
   server.start(SocketAddress("127.0.0.1", 8080));
}
```

The module descriptor is simple: I'm declaring version 1 of a module called `hello`, and I am importing the `ceylon.net`

module from the Ceylon SDK. The rest is the `run` function, which creates a web server with a single endpoint and starts it. The `shared` annotation means that this function is visible to clients that can view its container package (note that in Ceylon, functions can belong to packages, not just to classes and interfaces). The `value` keyword means "infer the type of that value by looking at the right side of the assignment." This greatly reduces verbosity while remaining statically typed.

Ceylon compiles to both Java bytecode and JavaScript. This multiplatform support enables you to run Ceylon programs not just on the JVMs you are used to, but also on browsers and Node.js virtual machines.

In Ceylon, functions can be invoked either with parentheses and ordered arguments, as in Java, or with curly braces, which allows you to specify arguments by name (`path` and `service`, in this code). In this case, the `service` argument is passed a trivial function that writes `hello world` to the client.

Something unique to Ceylon is that out of the box, the tools know how to compile and run this module. The IDE and even the command-line version know how to deal with modules; resolve and fetch dependencies; set up classpaths; compile and package modules—locally or even to remote module repositories; and all the tasks that other languages usually need many tools to perform. This is all you need to do on the command line to compile and run your module:

```
$ ceylon compile,run hello
```

The tools are smart enough that you don't need build tools for trivial matters.

When it comes to interoperability with other module systems, Ceylon modules include the necessary Maven and OSGi (Open Service Gateway initiative) metadata (soon `npm`, too). They even support the Java 9 Jigsaw project (although currently a flag is required to turn it on), because Ceylon already supports Java 9 modules. On JavaScript, Ceylon modules are compatible with the `Require.js` module system.

Thanks to modularity, the Ceylon distribution ships with only the `ceylon.language` module, which contains the basic Ceylon types; functions; and the metamodel, which is pretty small. Modules from the SDK are automatically obtained from online repositories when they are imported, and they are cached locally thereafter.

A Novel Friendly Type System

Ceylon features a powerful type system based on the following parts:

- Most types will be inferred (you've already seen this with the `value` declaration).
- For flow typing, once you have checked that a value is of a certain type or is not null, the type checker will remember it and allow you to treat it as such.
- The type of `null` and of object values should be distinct. An `Object` can never be `null`, and its type is distinct from `Null` (the type of `null`).
- Union types allow you to specify that a value should be of one of several types. For example, you can say that a value is of type `String` or of type `Null` with the `String|Null` union type. You can access members common to both types, or narrow the type to one of the cases with flow typing: `if(is String foo) then foo.lowercased else "<null>"`.
- Intersection types let you describe a value that should inherit from several types. An object of type `Appendable & Closeable` will be guaranteed to have all methods of each interface, no matter if it is a `File` or a `Logger`.

These features, along with some syntax sugar, allow you to

prevent `NullPointerExceptions` in Ceylon. If you have a value of type `String`, it cannot be null. If you have a value of type `String|Null` (sugared as `String?`), you cannot access its members until you have used flow typing to ascertain its existence. (In Ceylon, *existence* refers to whether a data item is non-null as determined by using the `exists` predicate.)

Optional Parameters

There are many great reasons to love Java. However, one feature I miss is the lack of optional parameters. Consider this Java code:

```
void log(String message){
   log(message, Priority.INFO);
}
void log(String message, Priority priority){
   log(message, priority, null);
}
void log(String message, Priority priority,
         Exception exception){
   ...
}

log("foo");
log("bar", debug);
// ouch, need to know the default
// value of "priority"
log("gee", info, x);
```

The corresponding code in Ceylon is this:

```
void log(String message, Priority priority = info,
         Exception? exception = null){
}

log("foo");
log("bar", debug);
log{
   exception = x;
   message = "gee";
};
```

Ceylon also does away with the verbosity of field and Java bean property accessors with a single attribute construct that can be final or dynamic:

```
class Counter(){
 // starts at zero, does not change on access
 shared variable Integer count = 0;
 // increments count every time it is accessed
 shared Integer increase => ++count;
}

// no need for "new":
// classes are functions in Ceylon
value counter = Counter();
print(counter.count); // 0
print(counter.increase); // 1
print(counter.count); // 1
```

There are many such examples of common Java itches that Ceylon scratches—and I haven't even talked about how awesome function types are.

The Ceylon SDK and Interoperability with Other Languages

Ceylon compiles to both Java bytecode and JavaScript (a Dart back end is nearing completion, too). This multiplatform support enables you to run Ceylon programs not just on the JVMs you are used to, but also on browsers and Node.js virtual machines (VMs). Once you have learned the Ceylon language (which was designed to be easy to learn), you can reuse that knowledge in your web applications on both the front and back ends.

Even better, you're not limited by what APIs you find in a particular back end. When running on the JVM, you have

excellent interoperability with every library out there: not just the JDK, but also every OSGi and Maven module. (Ceylon also knows how to resolve Maven modules from the Maven central repository.) People have run Ceylon code interacting with Spring, WildFly, Vert.x, and even Apache Spark (written in Scala). When compiled for the JVM, Ceylon produces idiomatic Java APIs that adhere to conventions (JavaBeans, for example) in most cases, which makes interoperability from Java to Ceylon easy.

When running on JavaScript VMs, you can easily access whatever APIs your browser provides, by using `dynamic` blocks, which relax the type checker to access untyped APIs. At the moment, the Ceylon team is working on `npm` integration (allowing you to use `npm` modules and publish to `npm` repositories) and on TypeScript interfacing, so that you can provide a typed interface to many JavaScript APIs, such as the DOM and browser APIs.

If you want to write libraries that work on both JVM and JavaScript VMs, though, interoperability is not always enough (although it is possible to write a module that delegates to different interoperability code depending on the back end). If you had to limit yourself to the JDK for collections, or to the corresponding Node.js module, you would never be able to write portable Ceylon modules. For that reason, Ceylon comes with a full-fledged modern SDK containing most of what you need to write portable Ceylon applications.

Ceylon runs out of the box on OpenShift, Vert.x (where you can even write your verticles in Ceylon), WildFly (produce a WAR file from your Ceylon modules with the `ceylon war` command), and OSGi environments, or simply from the `java` command line.

Additionally, because the Ceylon JVM back end is based on the Java compiler (javac), it can compile both Java and Ceylon source files at the same time, allowing parts of your project to be written in both languages.

Reified Generics

When using generics in languages that erase type-argument information at compile time (that is, languages such as Java without reified generics), it is often frustrating that all this information is lost at runtime. For example, you cannot use those type arguments anymore to reason about them. Without reified generics, you cannot ask if a value is of an argument's type, or use introspection on that type argument. For example, the following code can only be implemented in languages that have reified generics, such as Ceylon:

```
shared ObjectArray<T> toArray<T>(List<T> list){
    // if you have a List<Foo>, you create a
    //JVM array of Foo[], not just Object[]
    value ret = ObjectArray<T>(list.size);
    variable value x = 0;
    for(elem in list){
        ret.set(x++, elem);
    }
    return ret;
}
```

Nor is it possible to write the following without reified generics:

```
shared List<Subtype>
  narrow<Type,Subtype>(List<Type> origin)
    given SubType extends Type {

        value ret = ArrayList<Subtype>(origin.size);
        for(elem in origin){
          // Here we can check that the run-time
          // type is an instance of Subtype, which
          // is a type parameter
          if(is Subtype elem){
              ret.add(elem);
          }
    }
    return ret;
```

```
}
```

Or to write this more simply using a *comprehension*, which in Ceylon is a way to declare `Iterable` literals whose elements are generated by procedural code:

```
shared List<Subtype>
 narrow<Type,Subtype>(List<Type> origin)
   given SubType extends Type {
    // return an ArrayList whose elements
    // consist of every element of origin
    // that is an instance of Subtype
    return ArrayList<Subtype>{
      for (elem in origin)
        if (is Subtype elem)
            elem
    };
}
```

Metamodel

Ceylon also features a metamodel, which enables you to inspect all running modules, their packages, and their members (types, functions, and values), optionally filtered by annotation. This is very useful when creating frameworks. In order to point a framework to your packages, modules, or types, you can even use metamodel literals, which are similar to `Foo.class` literals in Java, except they can point to any element of the Ceylon program. For example, here is how you can scan the current package for Java Persistence API (JPA) entities:

```
shared SessionFactory
  createSessionFactory(Package modelPackage){
   value cfg = Configuration();
   for (klass in modelPackage
          .annotatedMembers<ClassDeclaration,
                            Entity>()) {
       cfg.addAnnotatedClass(
         javaClassFromDeclaration(klass));
   }
   return cfg.buildSessionFactory();
}

value s = createSessionFactory('package');
```

Good Tooling

Ceylon tooling does an excellent job, and many tools designed for Ceylon will make development easier. The first tool is the Ceylon IDE for Eclipse, which is a full-featured IDE for Ceylon with interoperability with Eclipse Java development tools. This IDE is what I use every day to develop Ceylon modules. It is very solid and supports more refactorings, quick fixes, and features than the Eclipse Java plugin. Naturally, like the rest of the Ceylon tools, it knows how to deal with modules and module repositories, because they are first-class citizens of the language.

In a few months, the Ceylon team will release the first version of the Ceylon IDE for IntelliJ IDEA. The team has been abstracting all the Eclipse plugin code into an IDE-agnostic module for the last few months, rewriting it in Ceylon at the same time, so that it can be used in both IntelliJ and Eclipse without having to maintain quick fixes and refactorings for both platforms. In the end, most of both IDEs will be written in Ceylon—a task for which the easy interoperability with Java helps a lot.

If you just want to try Ceylon without installing anything, there is a great Ceylon web IDE. It features several code examples and will let you write your own code, too, and it provides code completion, error markers, and documentation. So don't hesitate to throw your Ceylon code at it until you need a full-fledged IDE.

The other programmer's best friend is the command line. Ceylon has an extensive command line, inspired by Git, with a single `ceylon` command, from which you can dis-

cover every other subcommand via completion or via the `--help` argument. With it, you can compile and run on both back ends, document your modules, list available versions of a module, search for modules, copy modules or module repositories, and even package and install new command-line plugins.

For example, if you want to install and use the `ceylon format` command, just type:

```
$ ceylon plugin install ceylon.formatter/1.2.2
# now you can use it!
$ ceylon format ...
```

The Ceylon API documentation generator outputs beautiful, usable documentation, as you can observe from the language module documentation online. One of the features I like most is the ability to search and navigate results from the keyboard by typing `s`, searching, and then using the keyboard arrows. Try it!

The Ceylon distribution ships with Ant plugins for all standard command-line subcommands. Maven and Gradle plugins for Ceylon have been written by the community and are available, too, in case you want to include Ceylon as part of an existing Java application.

As your Ceylon modules mature, they can graduate from your computer to Herd: the online module repository for Ceylon. This open source web application functions as the central repository for every public Ceylon module. This is where you will find the Ceylon SDK, for example (it does not need to be bundled with the distribution). Anyone can have an account on Herd to publish modules, but if you want private deployments, you can also download the web application and run your own instance of it. By default, Ceylon tools attempt to fetch modules from the Herd repository if they cannot be found locally, but it's very easy to add other repositories.

Since the release of version 1.2.2, you can also ship a Ceylon autoinstaller with your projects. This installer uses the `ceylon bootstrap` command, which functions like the famous Gradle Wrapper, `gradlew`: it is a very small library that autoinstalls the right version of Ceylon for users who do not have it installed locally.

Conclusion

It is impossible to cover the entire language and ecosystem in one article. But if this introduction has made you curious, check out a good tour of Ceylon online.

Ceylon 1.0 was released two years ago. Ceylon is now at the mature version of 1.2.2, with a release cycle of around two or three months, to bring you bug fixes and features as fast as possible. The team has already merged two big feature branches bringing support for writing Android applications in Ceylon, as well as rebasing the JVM back end on the Java 8 compiler, which allows you to produce Java 8 bytecode on top of the existing Java 7 bytecode that Ceylon already supports.

Join the friendly Ceylon community online and feel free to post your questions. </article>

From deep in the mountains of Nice, France, **Stéphane Épardaud** (@UnFroMage) works for Red Hat on the Ceylon project, including the JVM compiler back end, various SDK modules, and the Herd module repository. He is a frequent speaker and is the co-lead of the Riviera Java User Group.

learn more

The official, in-depth tour of Ceylon

SIMON ROBERTS

Quiz Yourself

More questions from an author of the Java certification tests

Trusting that you're finding value in quiz questions with complete and detailed explanations of the answers, I've put together more interesting problems that simulate questions from the 1Z0-809 Programmer II exam.

Question 1. Given that the following methods are defined in an enclosing class called `Fruit`, which has the fields that support these method definitions, **which pair of `equals` and `hashCode` methods work together correctly to support the use of Fruit objects in collections?** Choose two.

Note: The `Objects` class is a core Java SE utility class that provides null-safe convenience methods that work as their names suggest.

a.
```
public boolean equals(Fruit other) {
  if (other == null) return false;
  if (!Objects.equals(name, other.name))
      return false;
  if (!Objects.equals(color, other.color))
      return false;
  return true;
}
```
b.
```
public final boolean equals(Object o) {
  if (o == null) return false;
  if (!(o instanceof Fruit)) return false;
  final Fruit other = (Fruit) o;
  if (!Objects.equals(name, other.name))
      return false;
  if (!Objects.equals(color, other.color))
      return false;
  return true;
}
```
c.
```
public long hashCode() {
  long hash = 7L;
  hash = 47 * hash + Objects.hashCode(name);
  hash = 47 * hash + Objects.hashCode(color);
  return hash;
}
```
d.
```
public int hashCode() {
  return Objects.hashCode(name);
}
```
e.
```
public int hashCode() {
  int hash = 7;
  hash = 47 * hash + Objects.hashCode(name);
  hash = 47 * hash + Objects.hashCode(color);
  hash = 47 * hash + this.weight;
  return hash;
}
```

Question 2. Given this:
```
Set<Fee> sf = new TreeSet<>();
```

Which of the following is required of `Fee` if the set is to behave properly? Choose one.

a. `Fee` must override the `equals` method of `Object`.
b. `Fee` must override the `hashCode` method of `Object`.
c. `Fee` must implement `Cloneable`.
d. `Fee` must implement `Comparable<Fee>`.
e. `Fee` must implement `Comparator<Fee>`.

Question 3. Given this code:

```
public class Goat {
  public static String getIt(String a, String b) {
    return "Hello " + a + " " + b;
  }
  public String getIt(Goat a, String b) {
    return "Goodbye " + a + " " + b;
  }
  public String getIt() { return "Goated! "; }
  public String getIt(String b) {
     return "Really " + b + "!"; }

  public static <E extends CharSequence>
    void show(BinaryOperator<E> op, E e1, E e2) {
    System.out.println("> " + op.apply(e1, e2));
  }
  public static <E extends Goat, F>
    void show(Function<E, F> op, E e, F f) {
    System.out.println(">> " + op.apply(e) + f);
  }
  public String toString() { return "Goat"; }

  public static void main(String[] args) {
    show(Goat::getIt, new Goat(), "baaa");
  }
}
```

What is the result? Choose one.

a. `> Hello Goat baaa`
b. `> Goodbye Goat baaa`
c. `>> Goodbye Goat baaa`
d. `>> Goated! baaa`
e. `>> Really baaa!`

Question 1. The correct answers are options B and D. This question delves into the meaning of equality and the nature of `hashCode`. Let's consider the matter of equality first. This seemingly simple notion can get quite complicated and troublesome in an object-oriented language.

First, let's find an `equals` method. To choose between options A and B, we can look at the argument types. The `equals` method is defined in the `Object` class, and the argument type is `Object`. An `equals` method such as is defined in option A, which takes another argument type, is an overload of the `equals` name, not an override of the `equals(Object)` method. Because of this, option A is incorrect, and option B must be the `equals` method we select.

Now let's consider the `hashCode` behavior that pairs correctly with this `equals` test. This method, also defined in `Object`, must return an `int`, and so we can reject option C, which returns a `long`. Now we have to choose between Options D and E. We can see that our chosen `equals` method (actually, either of them) tests the contents of `name` and `color` in determining equality, but the two `hashCode` methods we must choose between differ in this respect: option D considers only `name`, while option E considers `name`, `color`, and `weight`.

The requirement for a `hashCode` method is that if two objects, `a` and `b`, test as being equal using the `equals` method, then the `hashCode` values of each must be the same. That is, if `a.equals(b)`, then `a.hashCode() == b.hashCode()` must be true. Importantly, however, the inverse is not required. That is, just because two objects return false when tested with `equals` does not require different `hashCode`

values. Certainly it's better (from a performance perspective) if they are in fact different, but that's not required for correctness. It's interesting to note that it's both legal and functionally correct to define a `hashCode` method that simply returns the value 1. Doing so would result in the performance of all hash-based data structures containing such an object degenerating to the performance of something such as a linked list, but the containers would still work correctly.

So, our `hashCode` is not required to consider all the fields considered by the equality test, but it absolutely must not consider fields that are not part of the equality test. Because of this, we can reject option E, which considers the `weight` field, and we're left with the correct answer being option D, even though it might produce a suboptimal hash value.

By the way, failing to test all the relevant elements in a `hashCode` method is not just an esoteric laziness for the sake of asking "tricky" questions. The purpose of `hashCode` is to speed up searching, and sometimes a less choosy `hashCode` implementation that works faster might be preferable to a more discriminating method that takes longer to execute. In fact, the early versions of `java.lang.String` took exactly this approach.

Before we leave this question, let's consider a couple more points about the equality. Why did the equality test not have to consider `weight`, if that field exists? Where structured data types are concerned equality is, to some extent, a domain-specific concept. Imagine, for example, an implementation of a class similar in function to the `StringBuilder`. It's likely that this class would contain an array of characters holding the text and an integer indicating how many of the characters are part of the text. In this way, it's not necessary to resize the array every time the length of the text changes. But this also means that most of the time some of the characters in the array are not part of the text being represented, and they should not be considered in an equality comparison.

For example, consider two such structures. One initially contains `"Hello world!"` and the other contains `"Hello father"` and both have a length value of 12. Clearly at this point, they're not equal to one another. But if we truncate both to a length of 5, the character arrays still contain the same characters as before—that is, the two arrays contain dissimilar values—but the differences are irrelevant because the two objects now both represent `"Hello"` and should compare as equal.

From this, we can see that it's totally proper for an `equals` method to ignore some fields in an object.

In fact, equality testing is even more complicated when inheritance rears its head in object-oriented languages. Is a `TruckTire` of the same size "equal to" a regular `Tire`? All kinds of problems can arise here. Anyway, imagine that we have class `A`, that class `B` extends `A`, and that we have instances of these called `a` and `b`. Then if class `B` overrides the definition of `equals` in class `A`, `b.equals(a)` uses different behavior from `a.equals(b)`. That's very dangerous, because it allows the possibility that `b.equals(a)` could be false when `a.equals(b)` is true. That in turn breaks the reflexive property that is expected for equality and can result in very strange bugs. As a result, it's not unreasonable to define an `equals` method as `final`, even if the class itself is not.

For a very succinct listing of requirements, see the official documentation for `equals` and `hashCode`. If you want more discussion on these issues, with practical code-level advice on addressing the complex problems, take a look at the book *Effective Java* by Joshua Bloch.

Question 2. The correct answer is option D. This answer is perhaps a little surprising, but that's what makes it interesting. The basic behavior of a set is that it is an unordered collection that does not permit duplicates. The normal expectation in Java is that duplicates will be detected using the `equals` method of the objects, which suggests that the answer would be option A.

Meanwhile, a tree is a structure that depends on ordering. A set differs from a list in that the list should respect the order defined by the user of the list; but a set may use any internal order it likes, including one designed to allow finding items more quickly. That's exactly what a `TreeSet` does. Of course, if we're ordering the items to facilitate finding them, then some kind of ordering must exist for the objects in the set.

There are two ways this ordering can be provided. One option is that an external `Comparator` can be given to the `TreeSet` when it's created (which isn't an option for this question, because no such argument is provided in the setup code). The other option is that the object itself must implement `Comparable` so that it has what the Java documentation refers to as a "natural order." In this case, that means that option D is the correct answer.

Now, because the question permits only one answer, it might seem that we have a problem. Doesn't the `Fee` class also have to implement `equals` (and probably `hashCode`) to work properly? It turns out that the `compareTo` method, declared in the `Comparable` interface, can represent equality of a kind. The return from `compareTo` is an `int` value, with positive and negative results indicating that the argument value is ordered after or before the `this` value.

However, a return of zero from `compareTo` means that the two values being tested are at the same position in the ordering, and that's a form of equality. Indeed, if you think about this, you'll see that if the `compareTo` method says two objects are at the same point in the ordering, but they're not equal, we have a problem. We certainly don't know what to do with such values in a tree structure. To address the potential confusion in the general case, the documentation discusses the idea that a `compareTo` might be "consistent with equals" or not consistent (and strongly recommends that ordering be consistent with equals, but acknowledges that this is often impossible).

To resolve the ambiguity, the `TreeSet` documentation explicitly states that the `equals` method will not be used; the only comparisons will be done using the `compareTo` behavior. So, the only answer is in fact option D. `Clone`, of course, is a meaningless space filler in this case, because duplicating an object has nothing to do with being able to store it in a collection.

It's interesting to ask whether this would be a fair exam question. Should you have to learn this much about an API when you can look it up? To be fair, the exam writers probably would be a bit circumspect about a question that seems to depend on this much API detail (which is not to say such detail will not be asked, just that there won't be many questions depending on such knowledge). However, this particular answer can be worked out from an understanding of the meaning of a set, tree, and `Comparable`. You are expected to understand these things, and so answering the question does not, despite initial appearances, depend on rote learning, and is almost certainly valid. Here, I hope the question proved interesting and instructive. In a real exam, don't hesitate to use logic and understanding to eliminate items, or to determine that—as in this case—one particular answer is sufficient or "better."

Question 3. The correct answer is option D. There are two things that must be determined to decide how this code behaves. First, which `show` method will be executed? One of the methods takes three arguments that are constrained by the generics mechanism to be, for any given type variable `E` that extends `CharSequence`, a `BinaryOperator<E>` and two more parameters of the same type `E`. Because we call `show` with a method reference, a `Goat`, and a `String`, it should be clear that this cannot be the target method because `Goat` does not implement `CharSequence`.

The second `show` method also takes three arguments. The first is a `Function` that takes a `Goat`-like thing (that's the `<E extends Goat`... part) and returns something else. The second argument is also a `Goat`-like thing, and the

third is something else. This second `show` method then is the valid target of the call in the `main` method, and it tells us that the method reference must be formed into a `Function<Goat-like, String>`. That information tells us that the output must begin with `>>`, eliminating options A and B, and it is important for the next step in deducing the behavior, too.

So, how does the method reference get converted into a lambda that invokes one of the four `getIt` methods defined in this question? There are four distinct forms of method reference. These four forms invoke static methods, instance methods on objects defined in the method reference, instance methods on objects extracted from the lambda argument list, or a constructor. We can rule out the constructor version here, because the method reference uses the name `getIt`, not `new`.

If we have a method reference of the form `ClassName::methodName`, then it can create a lambda of the form `(a,b,c) -> d` if the class `ClassName` has a static method that takes three arguments that are compatible with the lambda's formal parameters (`a`, `b`, and `c`) and the static method has a return type that matches the return required for the lambda. In this question, we're trying to create a lambda that implements `Function<Goat-like, String>`. That would require a static method that takes a `Goat`-like argument and returns a `String`. The only static `getIt` method takes two `String` arguments, so it cannot be applied in this case. That allows us to reject option A (which, of course, we already rejected, because it has only a single `>` at the beginning).

If we had a method reference of the form `aGoatInstance::getIt`, we'd have an exactly parallel discussion about parameter mapping as for the static case, but searching for a compatible instance method. The method would be invoked on the instance `aGoatInstance`. However, that's not the form of the method reference, so we don't have to pursue that thought process.

The third possibility is that the method reference might translate into a lambda that's implemented using an instance method like this:

`SomeClass::aMethod` implementing `Function<A,B>` becomes `(a)->a.aMethod()`

In this situation, the input type to the function must be the type that precedes the double colon (`Goat`-like, in our question), and the return type of the method being referred to becomes the return type of the lambda (`String`). These types must be compatible with what the lambda must fulfill. In this case, we would need to make this translation:

`Goat::getIt` becomes `(Goat a) -> a.getIt()`

And the return type of `getIt` must be `String`. That's the one that returns `Goated!` and, therefore, option D is the correct answer. </article>

Simon Roberts joined Sun Microsystems in time to teach Sun's first Java classes in the UK. He created the Sun Certified Java Programmer and Sun Certified Java Developer exams. He wrote several Java certification guides and is currently a freelance educator who teaches at many large companies in Silicon Valley and around the world. He remains involved with Oracle's Java certification projects.

learn more

[Javadoc for Comparable]()

[Oracle tutorial on Comparable and comparators]()

[Introduction to Java 8 lambdas by Cay Horstmann]()

HARSHAD OAK

Getting Onboard Oracle Java Cloud Service

A hands-on, step-by-step guide to trying out an enterprise cloud

The Java cloud market is evolving quickly. Many vendors offer myriad cloud products and a lot of new terminology and jargon to go with them. In their early days, cloud solutions were mostly lightweight slivers of traditional server solutions. But today, we have pretty much all the functionality of large server-side products—often split into many specialized cloud solutions. This model comes with the benefit that developers are able to pick and choose enterprise options, and use only what's required.

Cloud solutions are commonly classified as follows:

- **IaaS.** Infrastructure as a service, or basic virtualized hardware and an operating system
- **PaaS.** Platform as a service, or IaaS with additional services, such as a database
- **SaaS.** Software as a service, or full applications on top of a PaaS stack

Within these categories, there are dozens of specialized cloud solutions. For example, Oracle offers many different cloud services in each of these categories. While at first this might appear overwhelming, the good news is that almost all cloud solutions are not "new" technology as such, which would require an understanding of new core technology. The challenge is more about getting accustomed to new interfaces, workflows, and terminology.

In this article, I explain what Oracle Java Cloud Service is and how to get onboard.

Oracle Java Cloud Service and Variants

Oracle Java Cloud Service began life a few years ago as a shared PaaS environment that offered support for commonly used Java EE technologies. Back then, it did not offer any fine control over the environment or the ability to tweak and customize based on requirements.

My previous Java cloud articles in *Java Magazine* ("Hands On with Oracle Java Cloud Service," September/October 2013, and "Build with NetBeans IDE, Deploy to Oracle Java Cloud Service," May/June 2014) discussed earlier versions of Oracle Java Cloud Service. Since then, Oracle has significantly enhanced its Oracle Java Cloud Service solutions.

Today, Oracle has the following three Oracle Java Cloud Service offerings:

Oracle Java Cloud Service - SaaS Extension. This is the Oracle Java Cloud Service offering that has been available the longest. It was renamed with *SaaS Extension* appended after the other two cloud services were launched.

As the name suggests, the primary use case this solution addresses is that of an Oracle SaaS user who needs to extend the capabilities of a SaaS offering. Because Oracle Java Cloud Service - SaaS Extension is primarily designed for this purpose, it offers easy integration with Oracle's SaaS solutions.

Note that although the name includes *SaaS Extension*, nothing in the product restricts you from deploying a standalone Java EE application that is not an extension of a SaaS cloud.

Oracle Java Cloud Service – SaaS Extension provides a shared PaaS environment where you can easily deploy Java EE applications without having to worry about any of the underlying hardware setup, server installations, patching, management, and more. Oracle Java Cloud Service – SaaS Extension supports all the commonly used Java EE technologies such as servlets, JavaServer Pages, JavaServer Faces, and Enterprise JavaBeans. It supports JAX-WS and REST web services. It also supports Oracle Application Development Framework, which is widely used among Oracle developers. It has most of the things Java EE applications require and is certainly a solution to consider when you are looking for a no-hassle, out-of-the-box shared PaaS environment for Java EE.

The primary differentiator for Oracle Java Cloud Service is that you can use the self-service portal to easily provision your environment to best suit your requirements.

Oracle Java Cloud Service – SaaS Extension does not let you configure the application server, the JVM, or the operating system to your exact requirements. This can work as an advantage in cases where the user does not want to be bothered by those things. However, in some enterprise applications, especially, greater control might be desired. Enter Oracle Java Cloud Service and Oracle Java Cloud Service – Virtual Image.

Oracle Java Cloud Service. The primary differentiator for Oracle Java Cloud Service is that you can use the self-service portal to easily provision your environment to best suit your requirements. You also have control of the underlying infrastructure and can choose Oracle WebLogic Server, memory, clustering, load balancing, virtual machines, and more. Setting up Oracle Java Cloud Service involves a lot more work and decision-making than Oracle Java Cloud Service – SaaS Extension, but you can get exactly what you require, and also have the freedom to further tweak things if needed in the future.

Oracle Java Cloud Service - Virtual Image. Oracle Java Cloud Service – Virtual Image is a similar environment to Oracle Java Cloud Service and also offers control over many aspects of the underlying environment. Oracle Java Cloud Service – Virtual Image is designed for use in development and testing, so it does not support backup and restoration, patching, or scaling. Setting it up is a little simpler because it has fewer prerequisites than does Oracle Java Cloud Service.

Note: The similar names of these cloud solutions can be somewhat confusing. So in the rest of the article, I treat these as three distinct products; notice carefully which one I am referring to in a particular context.

Getting Started with Oracle Java Cloud Service

Let's look at how to configure and provision a new Oracle Java Cloud Service instance. The first step is to request a free trial. Full-featured 30-day trials for Oracle Java Cloud Service and Oracle Java Cloud Service – SaaS Extension are currently available. I examine Oracle Java Cloud Service here.

Once your trial is approved, you fill out forms to set up your identity domain and login credentials. The identity domain is used to control the authentication, authorization, and features available to users. Users in an identity domain can be granted different levels of access to different cloud services. Once that has been set up, you can get down to provisioning the environment.

Log in to Oracle Cloud by entering your identity domain and login credentials. You will see a dashboard listing all services. As shown in Figure 1, you can use the drop-downs to show only particular services in a particular identity domain. In this case, I have marked a few services as favorites by clicking on the star icon and then only displayed those favorite services.

[Due to size constraints, other large images in this article are provided as links/downloads, which allows the images to be viewed at full size. —*Ed.*] Click the **Service Console** link for Oracle Java Cloud Service, and you will get to a welcome page for Oracle Java Cloud Service. Click the **Services** link (Figure 2) on that page to set up prerequisites.

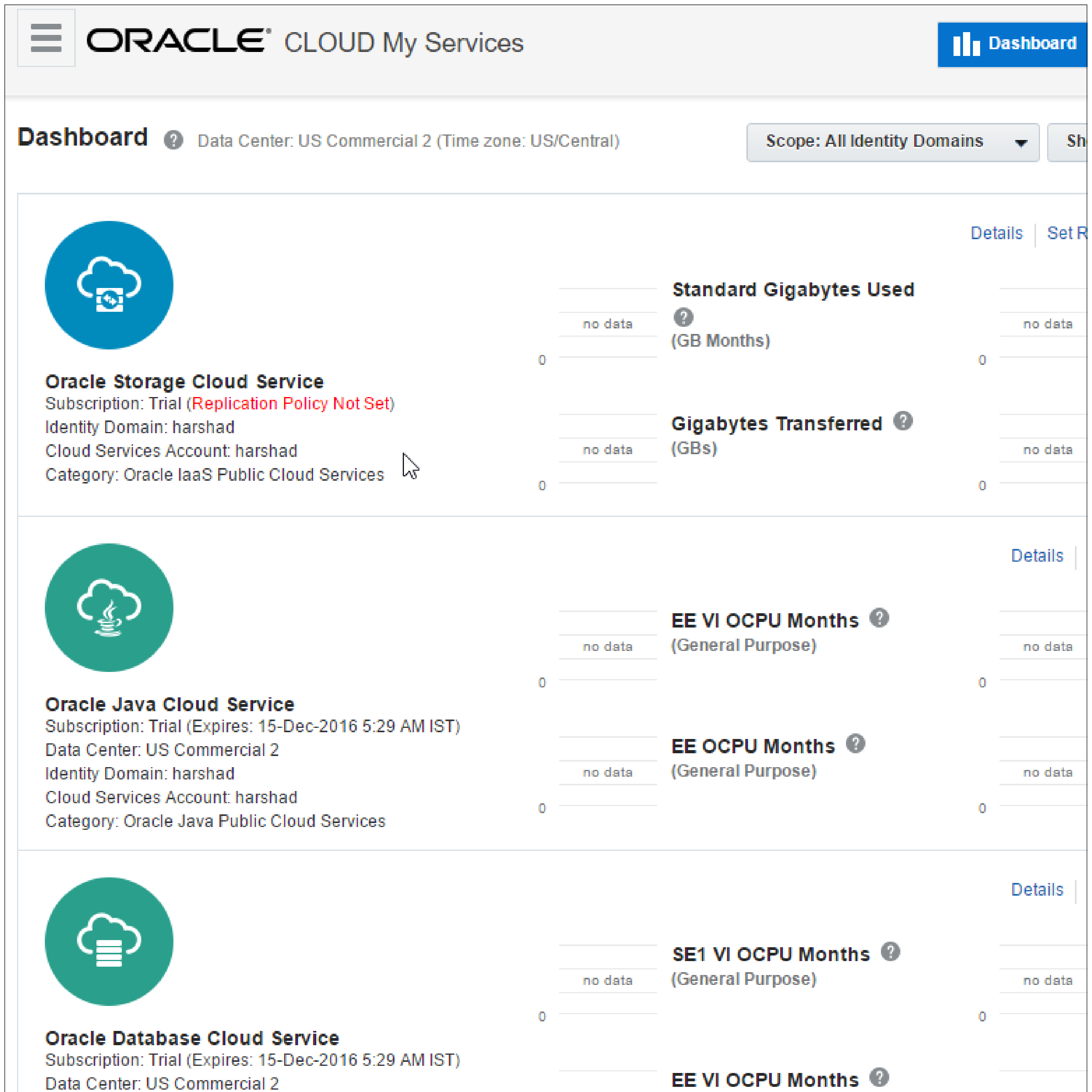


Figure 1. Configuration dashboard

The prerequisites are a Secure Shell (SSH) public/private key, an active Oracle Storage Cloud Service, and an active Oracle Database Cloud Service. The Oracle Java Cloud Service trial includes the trial versions of the other cloud services on which it depends, so you don't need to request any additional trials. Let's look at these prerequisites in more detail.

SSH Public/Private Key

Oracle Java Cloud Service requires an SSH public/private key pair for authenticating, so you need to generate one. I used the PuTTYgen tool (Windows .exe) to generate the key pair, but there are alternative ways as well. The public key is also required when provisioning Oracle Database Cloud Service and Oracle Java Cloud Service.

Oracle Storage Cloud Service

Oracle Storage Cloud Service offers a secure and scalable storage capability. Oracle Java Cloud Service requires Oracle Storage Cloud Service as it stores backups of service instances to a container in Oracle Storage Cloud Service.

You can see in Figure 1 that **Replication Policy Not Set** is highlighted against Oracle Storage Cloud Service. So first, you need to set a replication policy for Oracle Storage Cloud Service by clicking the **Set Replication Policy** link. For faster

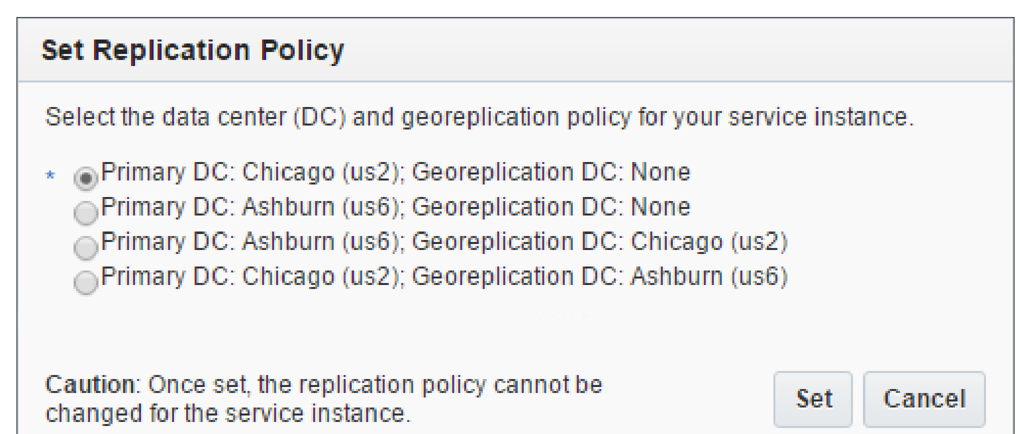


Figure 3. Storage replication policy

data transfers during replication, I recommend that you select the same primary data center to host the Oracle cloud services and Oracle Storage Cloud Service. Legal and security requirements also need to be considered.

As shown in Figure 3, I selected the same primary data center for Oracle Java Cloud Service.

Next, you need to create the required Oracle Storage Cloud Service containers for Oracle Java Cloud Service and Oracle Database Cloud Service. These containers can be created using the REST API or a Java library.

Note: If you are using the Virtual Image option of both Oracle Java Cloud Service and Oracle Database Cloud Service, you do not need to create the Oracle Storage Cloud Service containers. Because the Virtual Image is a development and testing environment, you have the option of not using Oracle Storage Cloud Service containers for backup and recovery.

Oracle Database Cloud Service

Oracle Java Cloud Service needs Oracle Database Cloud Service to be working. So, before you can create the Oracle Java Cloud Service instance, you need to first create the Oracle Database Cloud Service instance. Click the **Service Console** link for Oracle Database Cloud Service, as shown in Figure 1. On the following welcome page, click the **Services** link. You now get to the page shown in Figure 4.

Click **Create Service**. Next create Oracle Database Cloud Service by selecting the options for monthly billing and Oracle Database 12*c* Enterprise Edition on the **Service Details** page, as shown in Figure 5. I provided the service name `javamagDBWithStorage`, the description, and passwords. I also provided the SSH public key that I created earlier.

Using Oracle Java Cloud Service is simple enough and gets you scale and the other benefits that make the cloud a compelling proposition.

Note: If you are creating Oracle Database Cloud Service – Virtual Image, you can select the backup destination as none, so that you don't need to also set up an Oracle Storage Cloud Service container for Oracle Database Cloud Service backup and restore.

Select the basic shape with 1 OCPU and 7.5 GB RAM. The configuration can go up to 16 OCPUs and 240 GB RAM. (*OCPU* here stands for CPU capacity equivalent to one physical core of an indeterminate Intel Xeon processor with hyperthreading enabled.)

Click **Next** and confirm the details. In a few minutes, the `javamagDBWithStorage` database is provisioned and running, as shown in Figure 6.

Oracle Java Cloud Service Details

Once the `javamagDBWithStorage` database is up and running, head back to the Oracle Java Cloud Service console, as shown in Figure 2, and click **Create Service**. As shown in Figure 7, select Oracle Java Cloud Service. Then select the enterprise edition of the latest available version of Oracle WebLogic Server.

On the Service Details page, as shown in Figure 8, select the basic shape with 1 OCPU and 7.5 GB RAM, and specify the Oracle Database Cloud Service configuration and the Oracle Storage Cloud Service configuration for **Backup and Recovery Configuration**. Also specify the Oracle WebLogic Server username and password and choose to deploy a sample application.

Confirm the service information as shown in the summary in Figure 9. In a few minutes, the Oracle Java Cloud Service instance is provisioned and ready for use. Once that's done, you can use the instance of Oracle WebLogic Server similarly to an on-premises Oracle WebLogic Server. You can also log in to the Oracle WebLogic Administration Console to deploy applications to Oracle Java Cloud Service.

You now have Oracle Java Cloud Service set up, with an enterprise Oracle WebLogic Server and a database on the cloud. You also have backups and recovery set up on Oracle Storage Cloud Service.

Conclusion

As this article has shown, using Oracle Java Cloud Service is simple enough and gets you scale and the other benefits that make the cloud a compelling proposition, especially for enterprise applications.

The Java cloud space has matured rapidly over the past few years. In its early days, many developers had concerns: “Can the cloud be tweaked to get exactly what I want? Will the cloud bring all the power and functionality that I am used to getting from my on-premises server? Will it be flexible enough for my business?” And so on. In my experience, the newer Oracle Java Cloud Service solutions enable you to do all that and more. </article>

Harshad Oak is a Java Champion and the founder of IndicThreads and Rightrix Solutions. He is the author of *Pro Jakarta Commons* (Apress, 2004) and has written several books on Java EE. Oak has spoken at conferences in India, the United States, Sri Lanka, Thailand, and China.

learn more

[Oracle Developer Cloud Service]

[Oracle Managed Cloud Services]

Comments

We welcome your comments, corrections, opinions on topics we've covered, and any other thoughts you feel important to share with us or our readers. Unless you specifically tell us that your correspondence is private, we reserve the right to publish it in our Letters to the Editor section.

Article Proposals

We welcome article proposals on all topics regarding Java and other JVM languages, as well as the JVM itself. We also are interested in proposals for articles on Java utilities (either open source or those bundled with the JDK). Finally, algorithms, unusual but useful programming techniques, and most other topics that hard-core Java programmers would enjoy are of great interest to us, too. Please contact us with your ideas at javamag_us@oracle.com and we'll give you our thoughts on the topic and send you our nifty writer guidelines, which will give you more information on preparing an article.

Customer Service

If you're having trouble with your subscription, please contact the folks at java@halldata.com (phone +1.847.763.9635), who will do whatever they can to help.

Where?

Comments and article proposals should be sent to our editor, **Andrew Binstock**, at javamag_us@oracle.com.

While it will have no influence on our decision whether to publish your article or letter, cookies and edible treats will be gratefully accepted by our staff at *Java Magazine*, Oracle Corporation, 500 Oracle Parkway, MS OPL 3A, Redwood Shores, CA 94065, USA.

- Download area for code and other items
- *Java Magazine* in Japanese